AFRL-RY-WP-TR-2008-1228

# FUTURE FIELD PROGRAMMABLE GATE ARRAY (FPGA) DESIGN METHODOLOGIES AND TOOL FLOWS

**Dr. Michael Wirthlin, Dr. Brent Nelson, Dr. Brad Hutchings, Dr. Peter Athanas, and Dr. Shawn Bohner**

**Brigham Young University**

**JULY 2008**
**Final Report**

**AIR FORCE RESEARCH LABORATORY**
**SENSORS DIRECTORATE**
**WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320**
**AIR FORCE MATERIEL COMMAND**
**UNITED STATES AIR FORCE**

## NOTICE AND SIGNATURE PAGE

*//Signature//
ALFRED J. SCARPELLI
Project Engineer
Advanced Sensor Components Branch
Aerospace Components & Subsystems
Technology Division

//Signature//
BRADLEY J. PAUL, Chief
Chief, Advanced Sensor Components Branch
Aerospace Components & Subsystems
Technology Division
Sensors Directorate

//Signature//
WILLIAM J. SISKANINETZ
Chief, Aerospace Components & Subsystems
Technology Division
Sensors Directorate

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| **1. REPORT DATE** ***(DD-MM-YY)*** July 2008 | **2. REPORT TYPE** Final | **3. DATES COVERED** ***(From - To)*** 31 August 2007 – 31 July 2008 |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** FUTURE FIELD PROGRAMMABLE GATE ARRAY (FPGA) DESIGN METHODOLOGIES AND TOOL FLOWS | **5a. CONTRACT NUMBER** |
| | **5b. GRANT NUMBER** FA8650-07-C-7745 |
| | **5c. PROGRAM ELEMENT NUMBER** 69199F |
| **6. AUTHOR(S)** Dr. Michael Wirthlin, Dr. Brent Nelson, and Dr. Brad Hutchings (Brigham Young University) Dr. Peter Athanas and Dr. Shawn Bohner (Virginia Polytechnic Institute and State University) | **5d. PROJECT NUMBER** ARPS |
| | **5e. TASK NUMBER** ND |
| | **5f. WORK UNIT NUMBER** ARPSNDBR |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Brigham Young University, A-285 ASB, Provo, UT 84602; Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** Air Force Research Laboratory, Sensors Directorate, Wright-Patterson Air Force Base, OH 45433-7320, Air Force Materiel Command, United States Air Force; Defense Advanced Research Projects Agency/ Information Processing Techniques Office (DARPA/IPTO), 3701 N. Fairfax Drive, Arlington, VA 22203-1714 | **10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/RYDI |
| | **11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)** AFRL-RY-WP-TR-2008-1228 |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**13. SUPPLEMENTARY NOTES**
PAO Case Number: DARPA 12314; Clearance Date: 22 Oct 2008. This report contains color.

**14. ABSTRACT**
Interest is growing in the use of FPGA devices for high-performance, efficient parallel computation. The large amount of programmable logic, internal routing, and memory can be used to perform a wide variety of high-performance computation more efficiently than traditional microprocessor-based computing architectures. The productivity of FPGA design, however, is very low. FPGA design is very time consuming and requires low-level hardware design skills. This study investigated this FPGA design productivity problem and identified potential solutions that will provide revolutionary improvements in design productivity. Three research areas that must be addressed to achieve such improvements are significant improvement in reuse of FPGA circuits, identification and deployment of higher level design abstractions, and increasing the number of turns per day to significantly increase the number of design iterations. The results of this study suggest that with adequate advancement in each of these areas, FPGA design productivity can be increased by 25X over current practice.

**15. SUBJECT TERMS**
FPGA, design productivity, computer-aided design

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT:** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** (Monitor) |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | SAR | 60 | Alfred J. Scarpelli |
| Unclassified | Unclassified | Unclassified | | | **19b. TELEPHONE NUMBER** *(Include Area Code)* N/A |

**Standard Form 298 (Rev. 8-98)**
Prescribed by ANSI Std. Z39-18

# Table of Contents

## List of Figures

## List of Tables

## Acknowledgement

The authors gratefully acknowledge the support of DARPA/IPTO under contract FA8650-07-C-7745 and administered by AFRL/RYDI.

# 1 Executive Summary

The importance of Field Programmable Gate Arrays (FPGAs) for Department of Defense systems is well understood. The Special Technology Area Review (STAR) on FPGAs, for example, clearly indicates that FPGAs are a crucial electronic component in many DoD electronic systems (1). The report indicates that FPGAs will be used within many DoD systems for some time and will likely grow in importance as the performance and architectures of FPGAs improve. FPGAs are used within DoD for the same reasons they are used in commercial systems: reduced time to market, lower NRE costs, infield programmability, lower design and validation costs, and rapid prototyping. FPGAs also offer significant processing performance – by creating custom circuits optimized for a specific application, FPGAs can perform computations much more efficiently than other conventional forms of computing.

Several FPGA architecture trends suggest that FPGAs will become more important in the future. First, FPGAs are closely following Moore's law and are benefiting from the increased logic density available with new process technologies. Second, FPGAs are continually adding more system level functionality such as advanced I/O standards, bus interfaces, and memories. Third, FPGAs are integrating a variety of heterogeneous processing elements such as DSP processors, programmable processors, and computing elements. Fourth, FPGAs are providing multiple processors (both hard and soft) that can be organized into chip-level multiprocessing. This growing density, raw computational throughput, and system functionality suggests that FPGAs will play an increasingly important role in future DoD systems.

While FPGAs provide many benefits, the effort and skill required to create working FPGA designs is growing and consumes significant design resources during system development. The inability to create FPGA designs more productively limits the ability to exploit the growing density, capability, and performance potential of modern FPGA architectures. In fact, one of the key recommendations of the STAR report is the need to address the science and technology gap that includes the advancement of electronic design automation (EDA) for FPGAs. Unless significant advances in FPGA design productivity are made, the full benefits of FPGAs cannot be realized.

The objective of this effort was to investigate the *full* FPGA tool flow and identify potential solutions at all stages of the tool flow that will provide *revolutionary* improvements in design productivity. In the course of this study we have identified several key challenges limiting design productivity and identified several critical technical research focus areas to address the FPGA design productivity problem. This report summarizes our recommendations and proposes a research plan for solving the most important design productivity challenges. We believe that revolutionary advances can be made in FPGA design productivity with adequate investment in the research areas presented in this report.

The following section (Section 2) summarizes the background material and historical context for both FPGA design and software programming. Section 3 will introduce several metrics and present our "productivity model". This model will be used to identify the most promising approaches for improving design productivity. Section 4 will present the most promising approaches we have identified during the study that we believe will lead to revolutionary improvements in design productivity. Section 5 will conclude the report by presenting an integrated research vision that summarizes the vision from this study and the study conducted by the companion team made up from members of the National Science Foundation Center for High-Performance Reconfigurable Computing (CHREC).

# 2 Background

## 2.1 FPGA Devices

FPGA design productivity is limited by the so called design productivity gap (2). Silicon density continues to double every 1.5 to 2 years while design capabilities are growing at a much slower rate. Design productivity must improve at a rate similar to Moore's Law just to keep from falling behind. While incremental improvements in design productivity are being made, the rate of growth in design productivity is much lower than Moore’s law resulting in increasing design times for each new FPGA generation. Significant effort and investment in design techniques and methods are necessary for closing this design productivity gap.

Most of the largest FPGA devices available today are built using 65 nm technology[1]. These modern FPGAs contain a tremendous amount of logic, computation, and memory resources and can be used for a variety of high-speed digital systems and high-performance computing applications. The growth in density and capability of FPGAs will undoubtedly continue in the future. Table 1 suggests the resources that may become available on future FPGA devices using newer fabrication technologies. If FPGA density keeps pace with Moore's law, we expect the largest FPGAs in a 22 nm technology to contain almost 3 million look-up tables, several thousand dedicated multiplier/DSP blocks, and up to 100Mb of internal memory.

| Technology | Year | LUTs | DSPs | Memory |
|---|---|---|---|---|
| 65 nm | 2007 | 340 k | 500 | 10 Mbit |
| 45 nm | 2010 | 700 k | 1000 | 21 MBit |
| 32 nm | 2013 | 1,400 k | 2000 | 42 MBit |
| 22 nm | 2016 | 2,900 k | 4300 | 89 MBit |

**Table 1 - Density and Capability of Future FPGA Technologies**

While the density of future FPGAs will certainly increase, it is likely that the architecture of future FPGAs will continue to evolve. As more transistors become available, it is likely that the logic and computing resources will become coarser grain and more “hard-core” resources (such as PCI express) will be added to keep up with the latest and highest speed I/O interfaces. We also expect that a variety of new FPGA device families will be introduced to address the needs of specific markets. As such, FPGAs will present a moving target to Computer Aided Design (CAD) tools and we believe it will become increasingly difficult to address the gap between FPGA design productivity and FPGA circuit density.

## 2.2 FPGA Use Models

There has been considerable interest by non-traditional circuit designers to use and “program” FPGAs. These application experts and programmers recognize the benefits of FPGAs and seek ways to exploit the efficiency, reprogrammability, and computational density of FPGAs for their application-specific problems. These non-traditional FPGA programmers come from a variety of backgrounds including signal processing, embedded

[1] Altera announced the introduction of the first 40-nm FPGA (Stratix IV) on May 19, 2008.

systems, communications, and high-performance computing. These experts, however, do not have the traditional digital design skills to effectively "program" the FPGA using existing FPGA design tools.

The wide variety of users interested in using FPGAs suggests that new design methods and techniques are needed for FPGA design. We introduce the concept of an FPGA "use model" and define a number of "use models" to clarify the design issues that face FPGA designers and non-traditional FPGA programmers. Each model has a different set of design challenges, design constraints, and programming environments. While we have identified a variety of unique FPGA use models, we will focus on two FPGA use models for this report: *ASIC replacement* and *Configurable Computing*.

***ASIC Replacement*** is the most common FPGA use model. In this use model, FPGA devices are used to perform general purpose digital functions that might otherwise be performed in a custom integrated circuit (i.e., the FPGA is used to replace an ASIC). In this use model, the behavior and timing of the FPGA are specified in great detail including clock-cycle accuracy of the interfaces and internal logic. The design goal is to minimize cost (i.e., optimize hardware) and validate circuit functionality (including meeting timing constraints). The design is optimized in a way that allows the least expensive FPGA device to be used in the system. ASIC replacement applications typically involve the design of custom PC boards onto which the FPGA is placed, custom I/O interfaces, custom clocking requirements, etc. Much of the design activity involves creating the register transfer level implementation from some detailed system specification.

***Configurable computing*** is an FPGA use model in which FPGA devices are used to perform application specific *computation*. The large amount of logic resources available in modern FPGAs allows complex calculations and application-specific computations to be performed more efficiently and often with higher performance than more traditional CPU-based architectures (3). Standard platforms and boards are most often used for configurable computing to simplify the design process and facilitate reuse. When mapping a computation onto a configurable computing machine (CCM) the goal is often to get the design to fit into the available FPGA(s) as quickly as possible rather than to optimize the design down to the last gate.

The configurable computing use model has been applied in both high-performance computing (HPC) environments as well as high-performance *embedded* computing (HPEC). In both cases, FPGA designs are created on a standard platform to accelerate an application-specific computation. Unlike the FPGAs in an ASIC replacement use model, the FPGAs in configurable computing are reused for multiple computations. Because the FPGAs are reused and many FPGA designs created for a single design platform, design productivity is far more important for the configurable computing use model than for ASIC replacement.

Several ***emerging*** FPGA use models are being developed to facilitate the design of FPGAs in a variety of vertical markets. Many FPGAs are now used for Digital Signal Processing (DSP) and stream-based processing. A variety of new design methods are available for simplifying the design of FPGAs by DSP programmers (4). With embedded processor cores available within FPGAs, complex system-on-chip designs can be created within an FPGA. Design methods customized for SOC design have also been created for

FPGAs (5). Many other use models have been developed for a variety of application-specific tasks including networking (6), string matching (7) and many others.

A key reason design productivity for configurable computing is so poor is that that the design methods used in configurable computing are primarily the low-level design methods developed for the ASIC replacement use model. The design of configurable computing "programs" is essentially circuit design – low-level digital design methods such as RTL design are used to define complex computation and behavior. In fact, most of the design processes in contemporary configurable computing have direct counterparts in ASIC design (8). ASIC replacement design methods are insufficient for configurable computing and new methodologies are needed to improve design productivity. Development environments are needed for FPGA design that more closely resemble the development environments of traditional programmers and application developers.

While the development environments used by traditional programmers are varied, they possess a number of common traits. First, the languages used are abstract enough that a developer can create code with limited exposure to the underlying hardware structures. Second, developers expect a development environment consisting of compilers, extensive libraries of reusable functions, linkers, loaders, profilers, and symbolic debugging tools. Third, developers expect to work in an interactive development environment where the delay from compilation to debug on the target platform is measured in seconds or minutes, and the creation of *what-if* scenarios during the debug process is simple and efficient.

In contrast, development environments for FPGAs remain primitive by these standards. Developing for FPGAs currently requires detailed knowledge of the target chip's structure, capacity, and capabilities. Little in the way of reusable IP is available and logic analyzers and logic probes remain the key tools for the debug of most FPGA-based designs. Finally, FPGA development tool chains are batch-oriented rather than interactive with compile/link/execute timeframes measured in hours or days rather than seconds or minutes. Future advances in design productivity for FPGAs must significantly simplify the design/programming process of FPGAs for non-traditional FPGA users. In later sections of this report, our recommendations divide broadly into the three categories highlighted in the previous two paragraphs: abstraction, reuse, and development/debug environments.

We have focused our study on technologies and design methods that improve design productivity for *configurable computing* rather than for *ASIC replacement* or any of the other emerging use models. We believe that there is great potential for improving the design productivity for configurable computing and that with sufficient investment in a number of important technical areas, revolutionary improvements in design productivity for configurable computing are possible. While the techniques and ideas we present in this report are targeted towards configurable computing, we believe that many of these ideas can be successfully applied to the ASIC replacement use model and that some improvements in ASIC replacement design productivity are also possible.

## 2.3 Conventional FPGA Design Methodology

Before suggesting potential solutions to the FPGA design problem, it is useful to discuss the various phases of the conventional FPGA design methodology (i.e., design methodology used in the ASIC replacement use model). Furthermore, it is helpful to contrast these steps with the conventional software development process to highlight the added time, skill, and cost associated with FPGA design. Six broad design steps are highlighted in Figure 1 below and will be described in more detail.

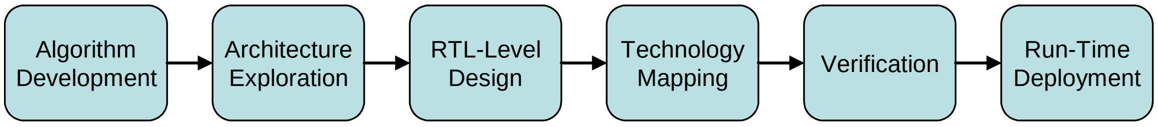

**Figure 1: FPGA Design Flow.**

### 2.3.1 Algorithm Development

Algorithm development is the process of creating and defining the behavior of the algorithm or computation that is intended for the FPGA. This is usually performed in a conventional programming language and tested using a variety of tools and software test benches. This step is common when targeting any computing platform including FPGAs, supercomputers, conventional microprocessors, etc. The focus of this step is to refine the algorithm rather than address implementation specific design details.

### 2.3.2 Architecture Exploration

Once an algorithm has been defined and verified, it must be targeted to a specific computing architecture. This task is broadly called architecture exploration and is unique for application-specific computing architectures including FPGAs. This step involves the creation of a unique, specialized computing architecture for the computation of interest. There is a very large design space for implementing these architectures and the primary challenge in this step is to identify the lowest cost architecture (size, power, etc.) that meets the computational constraints in as little time as possible. In most cases, this architecture exploration is performed manually by experienced design engineers[2]. This step is not necessary for software development as the hardware architecture is fixed.

### 2.3.3 Register Transfer Level (RTL) Design

Once an architecture has been identified for a computation, the architecture must be described using register transfer level design languages such as VHDL and Verilog. This process is not straight forward and requires the designer/programmer to explicitly schedule operations in time, allocate resources for these operations, and interconnect the resources. Further, the user must specify this architecture using hardware description languages that are unfamiliar to conventional programmers. While tools have recently been created that allow the description of these architectures in languages such as C, most of them require the programmer to be aware of architecture issues such as timing, parallelization, and resource allocation.

[2] Several high-level synthesis tools perform architecture exploration manually but these tools are not yet widely adopted by the FPGA design community.

### 2.3.4 Technology Mapping

After the design has been specified in a standard RTL-design language (or higher-level C-based language), it must be *mapped* onto the resources of a specific FPGA. This step is broadly called technology mapping and involves the *mapping* of logic to specific FPGA resources, the *placement* of these resources to specific locations within the device, the *routing* of signals between resources, and the generation of FPGA-specific programming bitfiles. Technology mapping is very time consuming – complex optimization algorithms are used to find acceptable logic placement and routing. As the size of FPGAs grows exponentially, the amount of time required for placement and routing grows significantly. An important limitation of FPGA design productivity is the long time required for place and route.

Unlike conventional software development, where compilation occurs in a matter of minutes, FPGA technology mapping may take many hours or days to complete for a complex design. As the density of FPGAs continues to grow exponentially, the time required for this technology mapping will grow to an unacceptable point. Technology mapping time must be reduced to improve FPGA design productivity for configurable computing systems.

### 2.3.5 Verification

After the computation has been mapped to an architecture and translated into an FPGA circuit, its proper functionality must be verified against the original algorithm description. Verification and debug is much more complicated on FPGA-based systems than conventional software because of the limited visibility within FPGAs, lack of control during execution, and the primitive interfaces and tools available for FPGA-based verification. If there are design errors within an FPGA-based computing system, it is significantly more difficult and time consuming to identify and resolve these problems than with conventional software tools.

### 2.3.6 Run-Time Support

The final step in the design and "deployment" of FPGA-based systems is providing appropriate run-time support. Unlike conventional processor-based architectures, there is limited support for the loading and managing of FPGA-based computations and interfacing these computations/architectures with conventional processor-based architectures. In most cases, ad-hoc or proprietary interfaces are used for each computing system adding significant time and cost to FPGA-based system design.

### 2.3.7 Detailed FPGA Design Flow

A more detailed diagram of the FPGA design flow is shown below in **Figure 2**. While the details of the design methodology are not important for this discussion, there are several observations that are worth emphasizing. First, there are many different activities required to create a valid FPGA design. These design steps require a variety of skills and tools to translate a high-level algorithm into a working FPGA system. FPGA designers must be skilled in each of these steps and tools to effectively create valid FPGA designs. Second, there are many feedback loops in the design process that require iteration, repair, and debugging. Iterations at all levels of the design flow are expected and multiply the amount of time required to create a valid design. Performing these design iterations significantly increases the overall FPGA design time.

### 2.3.8 Limitations of Existing Tools

Design tools for FPGAs continue to improve and provide the essential design support needed to create designs for today's large, complex, and heterogeneous FPGAs. These tools support the new features found in FPGA architectures and provide the capability to map complex designs to the largest available FPGAs. In addition, a variety of new design abstractions have been introduced to support new users of FPGA. These design abstractions include system on a chip design tools for embedded systems designers, signal flow graph tools for DSP engineers, and even C-based hardware compilers for algorithm experts.

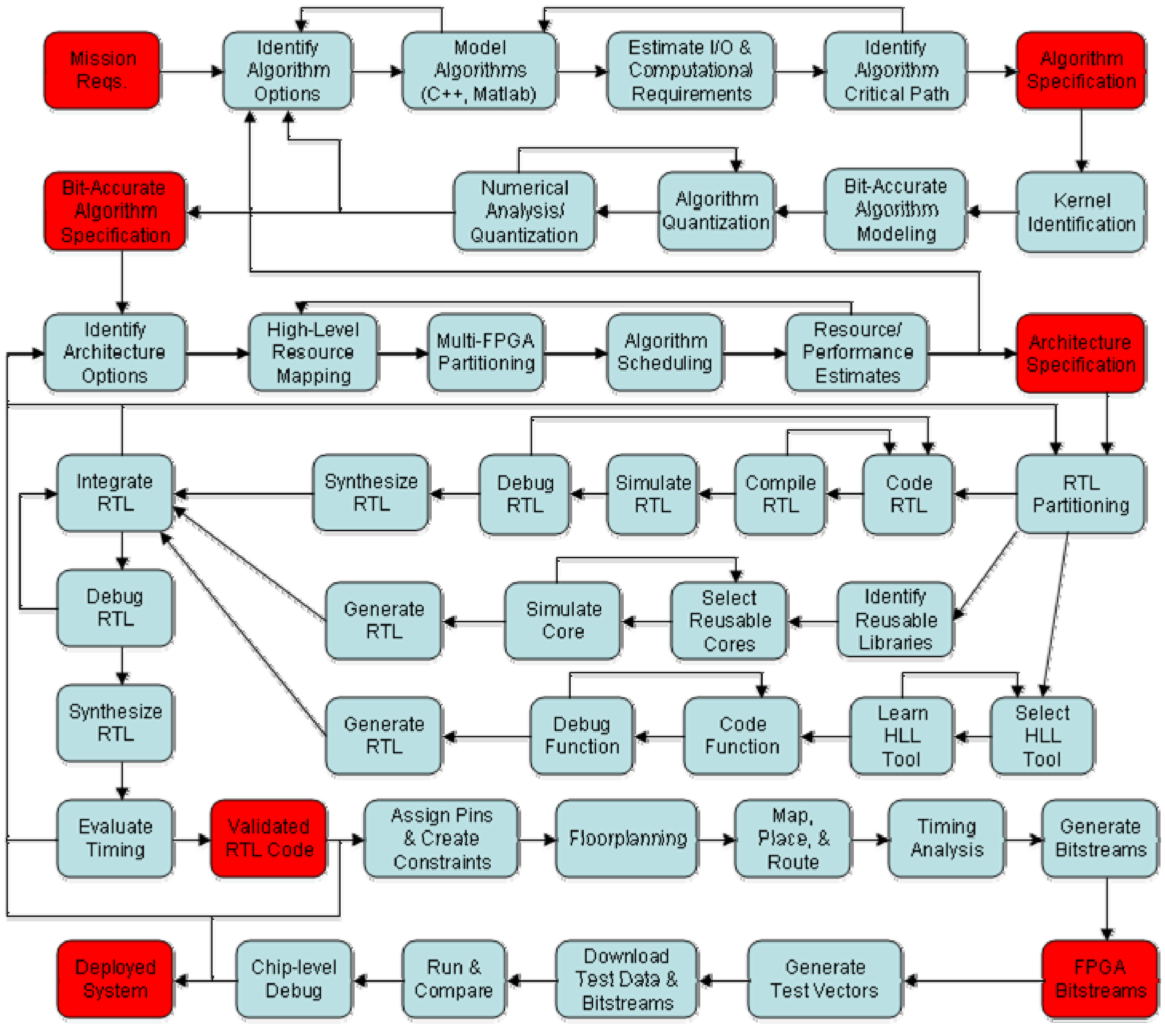

**Figure 2: Detailed FPGA Design Methodology.**

In spite of these improvements, FPGA designers frequently complain about the design tools. Improvements in FPGA design tools do not seem to keep up with the needs of the designers. The major limitations of the tools for traditional FPGA designers using FPGAs as an “ASIC replacement” include the following:

- Long place and route times,
- Difficulty meeting timing constraints,
- Difficulty verifying complex designs, and
- Inadequate design abstractions.

The tools for designers using FPGAs primarily for computation (i.e., the configurable computing use model) are primitive compared to traditional software development environments. As described earlier, these designers must use “ASIC” design tools to create computing circuits. There is a large mismatch between the background and skills of the algorithm expert and the current design entry tools required for FPGA design.

While new tools and abstractions for FPGAs are being introduced, these tools have not fundamentally changed the difficulty of FPGA design. In some cases, these new abstractions are not much different from traditional ASIC design and require the programmer to understand clocks, timing, and other low-level digital design concepts. In other cases, the abstractions are too restrictive and limit the ability of the synthesis tools to generate high-quality circuits (i.e., using sequential programming languages to specify concurrent hardware). In summary, the design of FPGA-based computing systems requires a variety of steps that each takes a large amount of time. Significant improvements in design productivity are only possible by addressing each of these steps and integrating these improvements into a cohesive design flow.

## 2.4 Historical Perspective

While current design methods for configurable computing closely resemble the design methods for ASIC replacement, the design goals and constraints of configurable computing are more closely related to traditional software development. In traditional software design, the programmer specifies high-level behavior and relies on optimizing compilers, profilers, debuggers, and other tools to automatically translate the behavioral description into an efficient implementation. Ideally, FPGA design for the configurable computing use model should look the same – programmers specify behavior in some high-level specification and use a variety of tools to translate this behavior into an efficient implementation onto the FPGA or configurable computing machine. Programmers should *not* be required to learn entirely new tool flows or become FPGA designers to successfully create FPGA circuits on reconfigurable platforms.

In the course of this study, the investigators regularly used software and the state-of-the-art in software productivity as the yardstick to measure various aspects of FPGA productivity. This was done for a few key reasons. First, there are many similarities between software development and FPGA design for computational problems. Since software environments are generally considered more mature than reconfigurable computing environments, this seems to be a good choice for longer-term trend analysis. Secondly, software productivity has progressed dramatically in nearly a half century. It would be a tremendous success if improvements in FPGA productivity could be aligned to the same productivity curves as software.

After reviewing the history of software productivity, the team noted that there have been three notable milestones, or *inflection points* in the course of software evolution that had significantly impacted software productivity.  These are:

1. The introduction of standard languages and compilers that promoted platform independence and code reuse (namely, the wide acceptance of FORTRAN and related languages).
2. The introduction of the linker, which in turn has lead to the preponderance of reusable code libraries.
3. Addressing human factors in software development by providing rich debugging environments and rapid turn-around for "what-if" development.

Computer programming started as a craft as computers became relevant in society in the 1960s.  Computer programming evolved into a science as more programming languages were developed for a variety of domain specific purposes.  In the 1980s it evolved into an engineering discipline as quality and scale became dominant issues. With each successive transition, productivity was improved.

Software productivity has increased steadily since the 1960s. Early on, micro-coding was the dominant programming approach. As more convenient machine (processor) structures emerged, assembly languages provided machine abstraction that improved productivity by over an order of magnitude. Then as programming domains such as business and scientific applications were established, third generation languages (3GL) like Cobol and Fortran with control and data flow abstractions led to another order of magnitude improvement in programmer productivity.

In 1970, COBOL was the state of the art, mainframes were in vogue and the personal computer had not hit the market.  By the early 1980s, it was clear that software productivity was a key bottleneck in many systems development efforts. In 1986, the Software Productivity Consortium (SPC) and the Software Engineering Institute (SEI) were formed to address the problem. Key areas like fourth generation languages (4GL) and fifth generation languages (5GL) were studied and some progress was made in specific domains where the workflow constructs could be aligned with computing capabilities. Much of the focus at these and other research organizations was on software reuse and integrated development environments. The SEI also started a program in software process that addressed process improvement.

Software environments also underwent a significant structural change since the 1960s.  In the 1960s, software tools focused on a model centered on the individual.  Code entry, compilation and debugging centered on the capabilities and limitations of individuals, and programming teams were comprised of individualistic effort.  Since then, there has been a major shift in this model to now focus on enterprise-level development with philosophical changes encompassing, code lifetime, reuse, verification and deployment (see Figure 3).  *Routine coding projects undertaken in today's software engineering environments could not have been accomplished using coding environments of the past.*

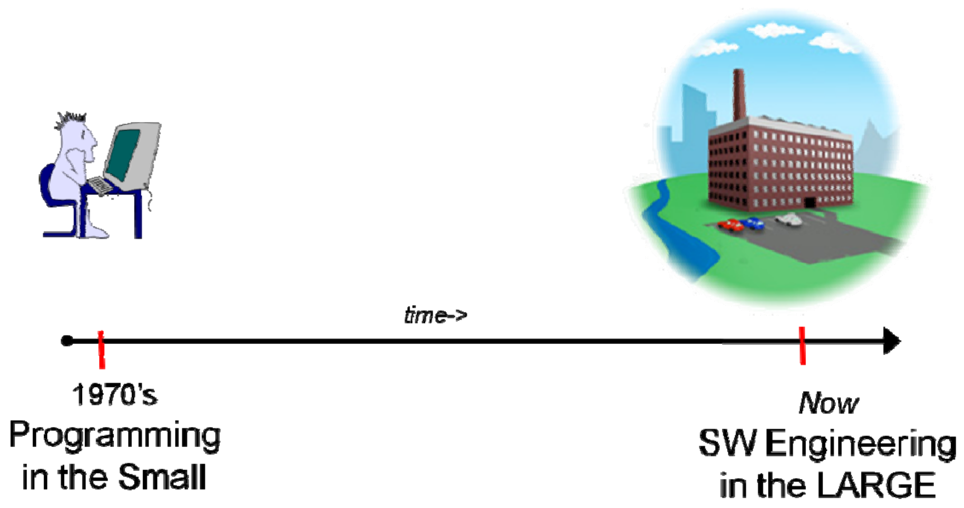

**Figure 3: The Fundamental Shift in Software Development Environments.**

Because of the close relationship between configurable computing design and software programming, it is instructive to look at the major innovations in software productivity over the last fifty years. *We believe that the current design tools and methods for configurable computing are still primitive and resemble the software practices of the 1960s.* Software productivity has progressed dramatically in the past half century and these improvements hold important insights for the configurable computing community. Many of the improvements in software productivity can be applied to configurable computing. The major advances in software productivity can be categorized into one of four different groups:

1. **Increased Abstraction**. Major improvements in programmer productivity have been realized by introducing new languages and design methods that reduce the amount of detail required by the programmer. The transition from machine code to assembly language and from assembly language to 3$^{rd}$ generation languages (9) allowed programmers to create complex programs without understanding low-level details of the microprocessor architecture.
2. **Reusable Artifacts**. An important way of improving software productivity is reusing previously created software artifacts (10). There are many levels of software reuse including reuse of applications, concepts, libraries, design patterns, and portable programs. The recent growth in reusable software components for web-based applications such as web services demonstrates the potential improvements in productivity through reuse.
3. **Software Process**. Recognizing that most early software development was done in an ad-hoc manner, new software processes were developed to improve productivity. Productivity improvements of 20% - 40% have been demonstrated for small software projects and up to 500% for large software projects (11) (12).
4. **Automation**. Automating tedious tasks played an important role in improving productivity (13). Tools to automate and integrate a variety of tasks have reduced errors and sped software development by over 30%.

As suggested above, configurable computing systems have yet to enjoy even the most basic productivity benefits demonstrated by software. While there are some encouraging signs of progress with new languages and compilation tools, contemporary FPGA design more closely resembles the lowest-level machine code programming of the very earliest computer systems. Significant advances in each of the four areas above are necessary for FPGA design in configurable computing systems to enjoy the benefits in productivity that were demonstrated by traditional software systems.

Using advances in software productivity as a guide, we have identified three broad technical areas that are most promising for configurable computing design productivity: *reusing artifacts*, *raising design abstractions*, and *increasing the interactivity and debug infrastructure* (i.e., "turns per day"). Software productivity has made *significant* advances in the last fifty years by making many advances in each of these areas. These areas of productivity are interrelated and design productivity will significantly increase if advances are made in each of these areas and applied at all levels of the design methodology.

# 3 Productivity Model

Before suggesting approaches and techniques for improving design productivity, we must have a clear definition and measure of design productivity. Closely related to the idea of design productivity are metrics for measuring design productivity. An appendix of this report (see Section 0) contains a sampling of papers we identified in the literature and which illustrate the state of the art in hardware design metrics. In essence, we found two kinds of hardware productivity metrics in the literature. The first and most common relates to input lines of source code created per day and is essentially an attempt to capture the amount of circuitry created per day. A second metric is the ratio of the utility of the system divided by its cost. While this latter metric is a more powerful metric and allows us to capture a variety of characteristics of the design process beyond simply circuitry created per day, we feel that the state-of-the-art in configurable computing design is such that we are not ready for this more complex metric, but prefer to use a simpler metric as a way of exposing what we view to be the most pressing problems in configurable computing design.

During the course of this study we developed a productivity model to guide our investigation (14). Models have limitations and the model we propose is no exception. It is not meant to predict the precise design time required for a given application or design. Rather, it is more qualitative in nature and points out what we believe to be the first-order contributors to design productivity and their inter-relationships.

Our first measure of design productivity is simply the rate at which hardware is developed:

$$DesignProductivity = \frac{CC}{DesignTime}. \quad (1)$$

Here, *CC* represents the *circuit complexity* of the final design, as measured in gates, LUTs, transistors, etc. The output of hardware design is hardware, a physical artifact that can be measured and that has quantifiable costs in several dimensions (silicon area, power, etc.). Unlike software, our model does not measure the *input* of the design process (i.e., lines of code/day) but rather the physical *output* of the design process (the amount of circuitry produced).

### 3.1.1 Design Time

The majority of the effort required to complete a hardware design is spent in debug and verification, with values in the 70% range being common. Thus, design time for configurable computing applications strongly depends on the number of design turns required to complete the verification of the design, and the ease with which those design turns can be completed. The design time is proportional to the number of design "turns" and can be approximated as:

$$Days = \frac{Turns}{TPD}, \quad (2)$$

where, *Turns* is the total number of design iterations required and *TPD* is "turns per day" (debug iterations per day).

### 3.1.2 Number of Turns Required to Complete a Design

The number of design turns required to generate a bug-free design (*Turns*) is dependent on the size of the input description as well as the frequency of occurrence of bugs embedded in that input description. We represent *Turns* as:

$$Turns = ILOC \times \frac{Bugs}{ILOC} \times \frac{Turns}{Bug}. \tag{3}$$

In this equation, *ILOC* stands for "Input Lines of Code" and should be considered as a proxy for the quantity "complexity of the design source", and could be measured in lines of input code, number of nodes in a graphical description of the circuit, etc.

The term *Bugs/ILOC* in Equation (3) is a measure of how many bugs are present per *ILOC* and is based on a simple assumption — that design errors are distributed uniformly through the design at a certain rate. Thus, the total number of bugs in a design is $ILOC \times {}^{Bugs}\!/\!_{ILOC}$. The assumption we make is that it takes one debug iteration (turn) to uncover and fix each bug. Thus, it can be seen that ${}^{Turns}\!/\!_{ILOC} = {}^{Bugs}\!/\!_{ILOC}$ and that ${}^{Turns}\!/\!_{Bug} = 1$, allowing us to rewrite equation (3) as:

$$Turns = ILOC \times \frac{Turns}{ILOC}. \tag{3b}$$

Combining Equations (1), (2), and (3b) leads to the following design productivity equation:

$$DesignProductivity = \frac{CC \times TPD}{ILOC \times \frac{Turns}{ILOC}}. \tag{4}$$

### 3.1.3 Effect of Reuse on Design Time

Equation (4) fails to capture the effect of reuse on design productivity. That is, design productivity improves when the designer is able to reuse pre-existing design pieces, requiring less original design. Reuse can be modeled as reducing the number of lines of code that the designer must write from scratch. *ILOC* (the code the user must create) can be modeled by two parts: first, the new portion of the design created from scratch and second, the interface code required to integrate the reused portions. It is useful to express this in a form where the amount of reuse is explicitly represented, along with the overhead associated with that reuse:

$$ILOC = ILOC_0 \times [(1-R) + (O \times R)]. \tag{5}$$

In this equation, $ILOC_0$ is the amount of code originally required to describe the circuit without the benefit of any reuse (the amount of code required to create it entirely from scratch). *R* is the *fraction* of the design satisfied by reusing circuitry – the user must only create $ILOC_0 \times (1 - R)$ lines of new design code.

Reuse is not free, however, and *O* represents the *overhead* of that reuse. It is expressed as a percentage of *R* and represents lines of new code that the designer must create to interface the reused circuitry to the rest of the design. As a concrete example, consider a design where $ILOC_0$=100, *R*=80%, and *O*=10%. Without the benefit of reuse, this would require the designer to write 100 lines of code. With reuse, the user would have to create: $100 \times [0.2 + 0.1 \times 0.8] = 28$ lines of code. The reuse overhead (*O*) reduces the benefit of reuse and if too high will eliminate any of the net advantages of reuse.

### 3.1.4 A Final Model

Substituting Equation (5) into Equation (4) gives the following final equation for design productivity:

$$DesignProductivity = \frac{CC \times TPD}{ILOC_0 \times [(1-R) + (O \times R)] \times \dfrac{Turns}{ILOC}} \ . \qquad (6)$$

This productivity model brings together design abstraction, turns per day, and reuse, and describes how each of these factors individually contributes to programmer productivity. We believe that orders of magnitude improvements in design productivity are possible if revolutionary advances are made in each of these three areas. For example, reuse alone may provide a 4× improvement in productivity as shown above. By developing and embracing higher levels of abstractions, the design detail required for a system may be reduced by a factor of 2× (i.e., increase the ratio of CC/ILOC by 2). Raising the abstraction and reusing FPGA artifacts may ultimately reduce the number of "turns" required to verify the design by 50% (Turns/ILOC). Finally, creating infrastructure, tools, and new processes to significantly improve interactivity may increase the "Turns per day" by 50% or more (i.e., 1.5× improvement). Taken together, these advances in all three areas would provide almost a 25× improvement in design productivity.

# 4 Research Approaches

The productivity model defined in the previous section identifies the research areas we feel are most important to address in order to substantially increase the design productivity of FPGA-based systems for configurable computing machines. These three research areas include reuse, raising design abstractions, and increasing the number of "turns per day". Each of these areas is interconnected and design productivity will significantly increase *only* if advances are made in each of these areas and applied at all levels of the design methodology

As described in the previous section, we believe that a 25× improvement in design productivity is possible (4× improvement due to reuse, 2× improvement due to higher level abstractions, and 3× improvement by increasing the number of turns per day). This section will describe several specific approaches that may lead to this 25× number. It is important to emphasize that no single technical advancement or approach will achieve this 25× productivity improvement. Advances in each of these three areas are necessary and at all levels of the design methodology. Further, the various advances that are made must interoperate and be integrated into a single design flow. If advances are not made in each of these areas or the advances are not mutually supportive, then much lower improvements in design productivity will be achieved.

This section will summarize each of the three research focus areas and provide specific approaches that we believe will achieve the 25× design productivity improvement. We believe these approaches are mutually compatible and necessary for achieving revolutionary improvements in design productivity. The approaches presented here are not exhaustive and we believe that there are likely other approaches that are compatible with this research agenda. We encourage the discussion and inclusion of other research approaches.

## *4.1 Reuse*

It is well known that reuse of software has been a significant factor in improving software design productivity (15) (16). Today's software systems are typically created by reusing software libraries, integrating reusable components, and dynamically integrating autonomous executables (COM, CORBA, etc.). Very large and complex software services can be created by exploiting the many available reusable software components and service oriented architectures. The successful exploitation of software reuse has led to significant improvements in productivity, higher quality code, fewer bugs, and lower software maintenance costs (17).

While these relatively new forms of reuse have provided remarkable improvements in productivity, software systems have exploited reuse of system infrastructure for many years. For example, even the simplest "Hello World" program involves a tremendous amount of code reuse. Reusable firmware, operating system calls, and run-time libraries are necessary to run this simple program. For example, consider the compilation of a simple hypothetical C program named "netmon.c":

```
gcc –o netmon netmon.c –lpthread –lm –lc
```

This program includes a variety of libraries and functions written by others to operate correctly. These reuseable libraries include:

- 285 functions in the C threads library,
- 400 functions in the C math library, and
- 2080 functions in the Standard C library.

The author of this simple program was not likely bothered with knowing the details of each library function or its interface, and could have developed the code on a different platform with a different processor. Despite this, the program likely produced the same results, differing possibly in temporal performance. The end result is that the amount of implicit and explicit reuse is immense in contemporary software practice.

Reuse within hardware systems, however, has significantly lagged behind that of software. While there is great interest in exploiting reuse for hardware design, the risk associated with reusing 3rd party circuits and the technical challenges of integrating "reusable" hardware circuits has inhibited the widespread adoption of reuse methods. One study suggested that if the time required to reuse a component was greater than 30% of the time required to design the component from scratch, design reuse would fail (designers would choose not to reuse) (18). The risk and cost of hardware reuse must be reduced before hardware reuse is widely used.

While hardware reuse is difficult, the potential improvements in productivity are significant (19). For example, if 80% of a hardware design is created by reusing existing hardware (i.e., $R=0.8$) and the effort to integrate reusable hardware is 10% (i.e., $O=0.1$) then hardware design productivity will increase by a factor of 4 (see Equation (6)). Achieving this level of reuse today and at such a low cost is difficult. However, the improvements in software reuse over the last four decades suggests that significant improvements in hardware reuse can be made with appropriate technology advancement and community cooperation.

There are other side benefits of increased reuse in a hardware development environment beyond library elements and core sharing. Attaining a degree of design mobility is important as new technologies are introduced (Figure 4a), and existing designs age and become unusable *legacy code* (Figure 4b). Like software, there are many different ways to exploit reuse during the design and deployment of a hardware system. These include the following:

- Library cell reuse - this is what most people think of when reuse is proposed and is the use of cells from a standard library which perform a specific function (FFT, for example).
- Retargeting reuse - the porting of designs between devices from different manufacturers or even between devices from a single manufacturer.
- Design pattern reuse - the reuse of structures such as pipelining or bit-serial arithmetic in the creation of a design (20).
- Architecture reuse - meta-architectures are architectures layered on top of traditional reconfigurable fabrics to facilitate reuse.
- Platform reuse - the use of standard CCM-like platforms with FPGAs, memories, and I/O capabilities.

- Interface reuse - the use of standard I/O connections to alleviate the designer creating custom interconnect for each application.
- Technology mapping reuse – the reuse of place and route information on circuit components that do not change.

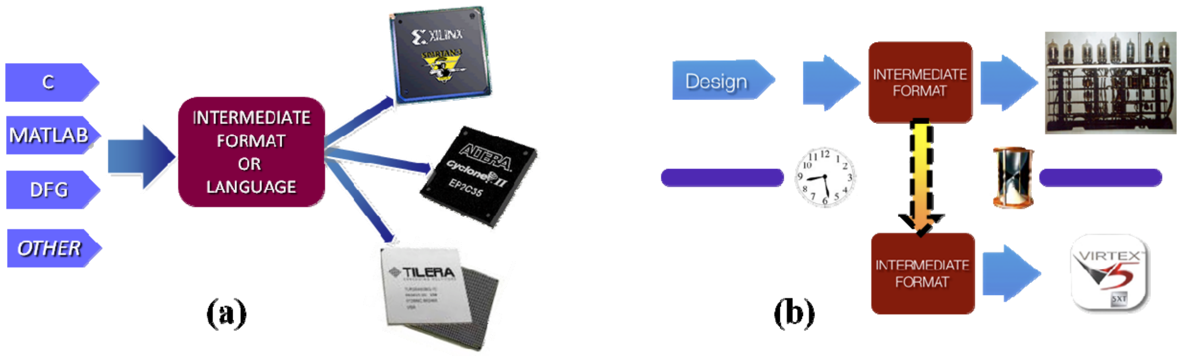

**Figure 4: Two Key Benefits of Hardware Reuse: (a) The Ability to Retarget other Devices, and (b) Mitigation of Obsolescence.**

We propose four specific research topics related to reuse that we believe can significantly improve the benefits of reuse within the FPGA design flow.

### 4.1.1 Library Reuse Infrastructure

The most common and direct form of hardware reuse is the reuse of hardware components. Predefined hardware circuits (otherwise known as "intellectual property" or IP cores) are created and verified and then later inserted in a larger hardware circuit. While such reuse occurs frequently within an organization, reuse between organizations and third-party developers is limited. In addition, it is difficult to reuse hardware components over time – they become obsolete and reusing today's modules on tomorrow's devices is problematic.

One problem is the lack of standards – hardware circuits are developed in a variety of tools and incompatible languages that inhibit the reuse of the circuit in new environments and design flows. Developing standards for describing and representing reusable hardware will enable a variety of high-level tools to take advantage of a variety of cell libraries developed within different tools (21). Figure 5 demonstrates how a common standard for libraries can significantly improve reuse. A common standard for representing circuit libraries and cores will allow any core using this standard to be seamlessly integrated to any high-level tool.

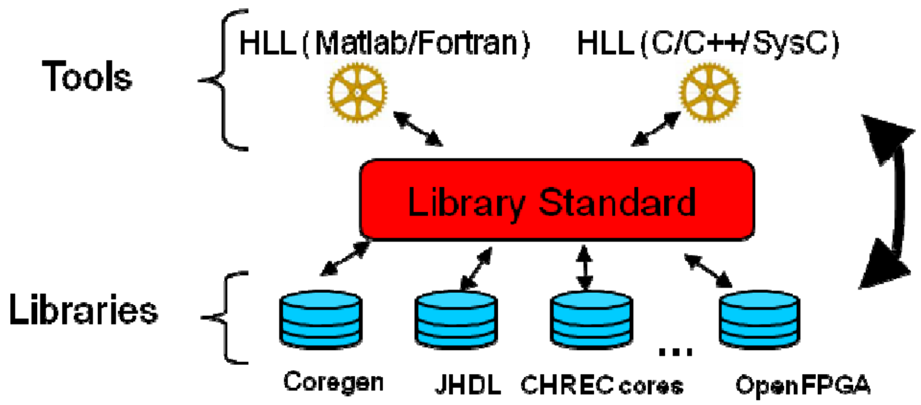

**Figure 5: Library Standard for Reusable FPGA Libraries.**

The concept of library reuse could go one step further and adopt the library and sharing models that have demonstrated promise in the software engineering realm. One example from the software realm is the Common Object Request Broker Architecture (CORBA), which enables software components written in multiple computer languages and running on multiple computers to work together. This objective is similar to the needs of reconfigurable computing, but goes one step further (see Figure 6).

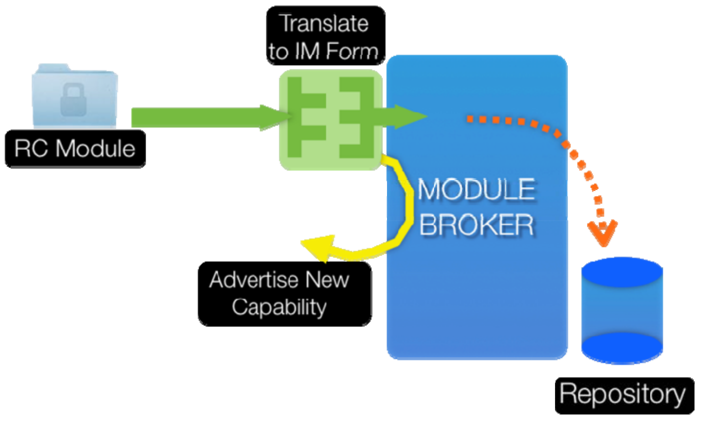

**Figure 6: CORBA-Like Flow for Reconfigurable Computing.**

In reconfigurable computing, a repository architecture is desired that not only enables hardware components written using different specification languages to be maintained in a common repository, but also provides the capability of interface synthesis (see Section 4.1.4) that promotes IP portability. A use-model of this concept is as follows:

- A standard is established for describing core interfaces,
- Reusable cores are cataloged within the standard,
- Tools automatically import core using core description,
- Tool or designer requests information about cores, and
- A "push" model can be developed where core capabilities and interfaces are advertised by the repository.

In its most refined form, a compilation tool would be aware of advertised capabilities, perform the necessary trade-off analysis, select the appropriate core, and synthesize its interface *automatically*.

It is important to emphasize that the process of creating a library of reusable components is only half of the picture. Performing operations on this library and making it easily accessible is the other half. By reducing the component search time and promoting integration, library extensions such as this would have an obvious impact in enhancing reuse in a typical design environment, leading to a doubling in productivity.

Implementing this concept is not simply a software development task – there a variety of difficult issues and questions that must be resolved before any standard or library infrastructure could be developed. Difficult questions that must be addressed include the following:

- What is the essential information necessary to represent a reusable core?
- How do you represent details of a low-level core at multiple levels of abstraction?
- How do you integrate the module generators and other core library infrastructure to high level tools?
- How do you advertise the capabilities, options, and performance of a core?

We believe that when these questions are properly answered and standards are created that address these issues, it will be significantly easier to reuse circuit libraries leading to notable improvements in design productivity.

### 4.1.2 Architecture Shaping Through Library Standards

Standardized well-characterized libraries, common among all qualified DoD FPGA vendors, would greatly enhance code reuse and code portability and mitigate early obsolescence of code bases. In the software world, standardized libraries such as VSIPL (22) and LinPack have directly affected how compilers are built and even how machines are made. If such a configurable computing library had a (forcibly) high adoption rate, it is likely that device vendors would be motivated to optimize their mappings to elements in the library, or even make architectural enhancements to give them a competitive advantage over their peers. This seems to be an obvious tactic for the industry to deploy; however, there is currently little incentive for FPGA vendors to do this. Furthermore, contemporary FPGA architectures are crafted to suit the needs of their primary customers who value logic density above all else. It is conceivable that a critical mass of users with a common use-model (via mandatory library interfaces) could ultimately inspire competitive forces among device manufactures to optimize their architectures. This process is referred to here as architecture shaping, and is accomplished through the following four steps:

**STEP-1**: Create a consortium for the purpose of defining (domain-specific) reconfigurable computing libraries and standards. This will likely need to be a grass-roots endeavor since widespread adoption of the library is important. Unlike traditional core libraries, this would need to capture non-traditional building-blocks, such as a class of elements devoted towards connectivity and data movement.

**STEP-2**: Once there is established widespread acceptance of the standard and constituent libraries (either through perceived convenience, productivity benefits, or even coercion), there would be natural forces from vendors and users to create efficient mappings to devices.

**STEP-3**: Once there is reasonable acceptance of the standard, and that there is a means of mapping designs to the standard, the DoD could then mandate that all reconfigurable computing designs be expressed in the standard. This would be similar to the mandate that arose in the VHSIC program in regards to the usage of VHDL in DoD designs.

**STEP-4**: At this point, designers will be less driven by particular vendors for their design implementations, and more driven by libraries and standards. This achieves a degree of vendor independence for the designer, and all of the other advantages that come with it including design mobility, second source satisfaction, and economy-in-scale. Vendors in turn will need to demonstrate a competitive advantage. As vendors compete, they will develop highly tuned implementations and possibly enhance their architectures. Vendor A could claim an advantage if they were to produce an enhancement to their device that more efficiently mapped standard library primitives.

There is historical precedent that suggests that FPGA architecture shaping can achieve success. Consider the RISC "revolution" of the 1980s. Here, the concept of highly dense and complex ISAs (analogous to contemporary hardware-centric FPGA architectures) were abandoned in favor of giving the compiler more control in the process. If there were an entity that could create a broadly acceptable library, possibly through a standards process, it is possible that a "critical mass" could be attained. Compliance to this standard could be mandated by the DoD as a condition of these requirements and mandates could be phased in over time. Ultimately, vendors could be required to comply as a condition for DoD participation.

There are potentially secondary rewards from architecture shaping as shown in Figure 7. Standards will also create the opportunity for 3rd-party tool vendors to compete in the CAD space that is currently mostly exclusive to the device vendors. This could potentially impact the TPD factor in the productivity equation.

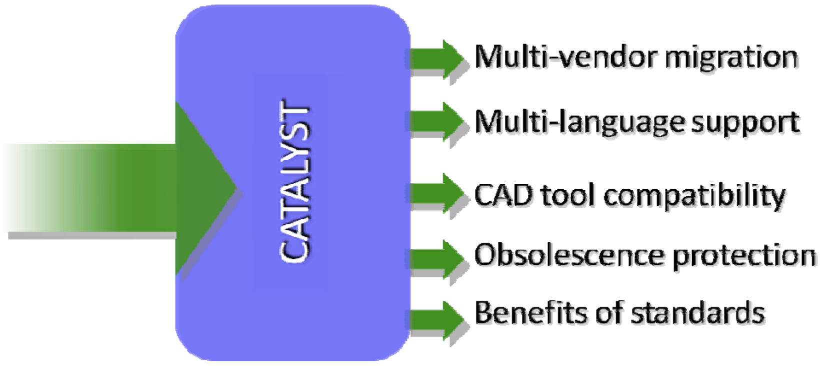

**Figure 7: Catalytic Impact of Architecture Shaping and Leveraging Library Standards.**

### 4.1.3 Dual Layer Compilation

Synthesizing computing circuits onto arbitrary hardware is much more difficult than compiling a program onto a sequential processor. Computing tasks and memory accesses must be assigned to resources and scheduled in time. A two-level compilation strategy may assist the compiler and synthesis tools during this process. Standard "meta-architectures" are defined that represent more coarse grain architectures than FPGAs and provide a higher level abstraction than low-level LUTs and wires (23). The compilation and synthesis process can be simplified by compiling to this meta architecture level using higher level abstraction tools and then using low-level device specific tools to generate actual computing circuits. Further, a two-level compilation strategy will lead to greater portability and reusability by more easily allowing computations compiled to a meta-architecture to be retargeted to other low-level device architectures.

One notable outcome of the DARPA Polymorphous Computing Architectures (PCA) program was that concept of dual-layer compilation. Briefly, the PCA dual layer approach decomposed the compilation process into (1) a stable API layer, responsible for transforming a variety of standard programming languages into a common intermediate format, and (2) a stable architecture abstraction layer, that transformed the intermediate layer into a form amenable to the target hardware (23). While the original motivation behind this concept is somewhat different than the motivation for FPGA productivity, both share many of the same properties in that:

- The dual-layer process is open to a wide variety of input specification languages.
- The dual-layer process does not change the familiar coding environment expected by the designer.
- If designed appropriately, little efficiency is lost when working in an intermediate architecture abstraction layer.
- Vendor specific back-ends can be developed independently (by the device vendors), gaining the ability to retarget different devices.

Overall, the impact on productivity by adopting this approach could be large: reuse is improved by intentionally separating the language problem, and the device-mapping problem. Much planning would need to go into the design of the architecture abstraction layer to preserve mapping efficiency. The Reservoir Lab *R-Stream* project, summarized in Figure 8, has many of the salient features that could benefit reconfigurable computing. Here, a problem is described in a high-level language, and compiled into a "Virtual Machine Abstraction" intermediate form. This can in turn also be a C specification, but transformed in a way in which the optimization dimensions are exposed. At this point, device-specific compilers can then be used to create the target image. For example, Xilinx's CHIMPS could be use to compile the *low-level C* (LLC) into an FPGA bitstream, or a version of NVidia's CUDA compiler could transform the same LLC into something suitable for a GPU.

While the multitude of C-to-Gates compiler efforts have incrementally improved over the past 20 years, they have not come close to closing the productivity gap, and there is no revolutionary change envisioned that is likely to change this. Furthermore, parallel programming languages that emphasize letting the user adjust aspects of the mapping process within the language are likely critically flawed. While they may seem to initially promote productivity, they in effect anchor the design to a particular technology, and possibly a particular platform. There have been notable attempts in the past, that have shown that the added constructs distract the programmer from focusing on the problem space to mixing physical implementation issues in the specification. The result is a set of tools that are non-portable and non-compatible.

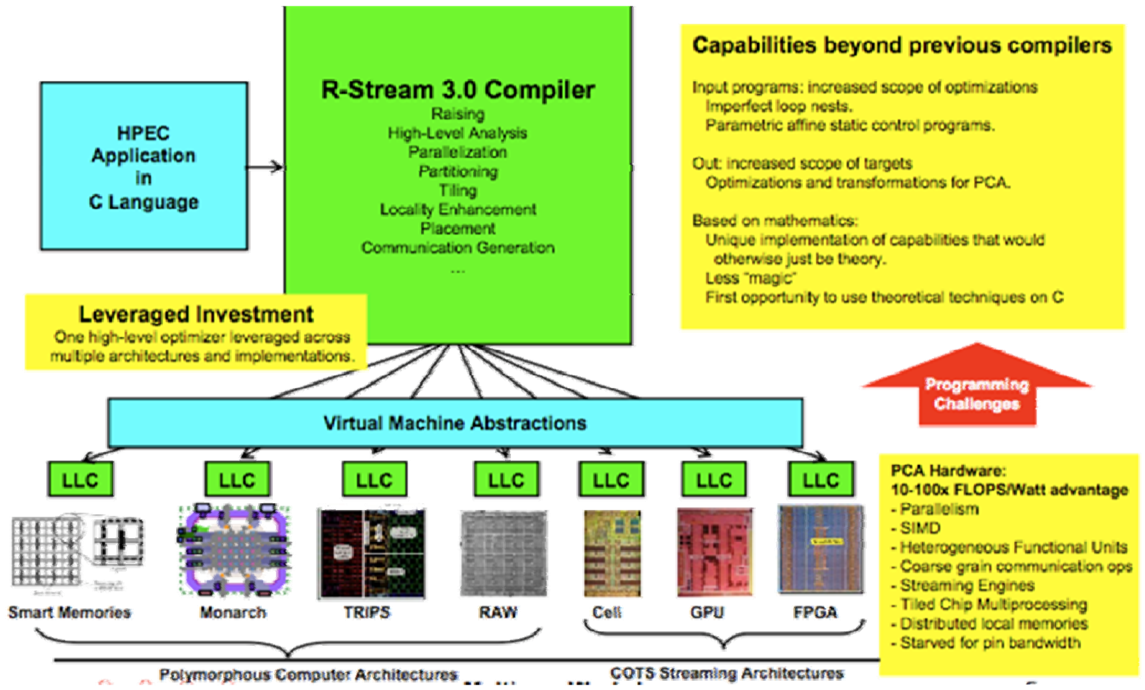
HPEC Application in C Language
R-Stream 3.0 Compiler
Raising
High-Level Analysis
Parallelization
Partitioning
Tiling
Locality Enhancement
Placement
Communication Generation
...
Capabilities beyond previous compilers
Input programs: increased scope of optimizations
Imperfect loop nests.
Parametric affine static control programs.
Out: increased scope of targets
Optimizations and transformations for PCA.
Based on mathematics:
Unique implementation of capabilities that would otherwise just be theory.
Less "magic"
First opportunity to use theoretical techniques on C
Leveraged Investment
One high-level optimizer leveraged across multiple architectures and implementations.
Programming Challenges
Virtual Machine Abstractions
LLC
Smart Memories
Monarch
TRIPS
RAW
Cell
GPU
FPGA
PCA Hardware:
10-100x FLOPS/Watt advantage
- Parallelism
- SIMD
- Heterogeneous Functional Units
- Coarse grain communication ops
- Streaming Engines
- Tiled Chip Multiprocessing
- Distributed local memories
- Starved for pin bandwidth
Polymorphous Computer Architectures
COTS Streaming Architectures

**Figure 8: An Outline of the Dual-Layer Compilation Work of the Reservoir Labs R-Stream Project.**

### 4.1.4 Interface Synthesis

FPGA circuits are difficult to reuse for several reasons. First, the designer must choose a circuit to reuse. There are a wide variety of cores and libraries that vary in many parameters (speed, area, power, etc.). It can be time consuming to search through the available cores and select an appropriate reusable circuit. Second, the designer must understand the low-level details of the reusable circuit interface. This may involve reading the low level HDL code or reading detailed documentation. Third, the designer must create custom circuits to talk to the interface, and fourth, the designer must then verify the system with the reusable core. Much of the time involved in reusing FPGA circuits is the extra design time required to interface a reusable circuit to a new system (see Figure 9). Unless this additional "reuse" time is significantly reduced, the improvements in productivity due to reuse will be limited.

**Figure 9: The Primary Challenge of Integrating Reusable Components is Creating a Custom Interface.**

As noted in our productivity model, reuse does not come for free, where there is typically a cost-benefit trade-off associated with it.  It has been noted in the literature that circuit designers are reluctant to reuse circuits unless reuse integration costs are less than 30% of the original design time. Therefore, an essential aspect of reuse is making the usage of a reusable component easy.

The objective of interface synthesis is to reduce the effort required to reuse a circuit. This is possible by automatically synthesizing the interface between a reusable circuit and the new circuit (see Figure 10).  Interface synthesis is done by encapsulating the circuit interface of reusable circuits in meta-data descriptions and automatically synthesizing the interface between the circuit and the system.  If done properly, modules can "seamlessly" transition from one design with one set of interface requirements and standards to another design.  The use-model for interface synthesis is straightforward.  First, it assumed that the circuit interfaces are created (preferably with a degree of automation), and are specified by meta-data.  This provides sufficient information for the compiler to synthesize circuit-specific interface logic.  In the user's perspective, reusable circuits are integrated with little or no effort.

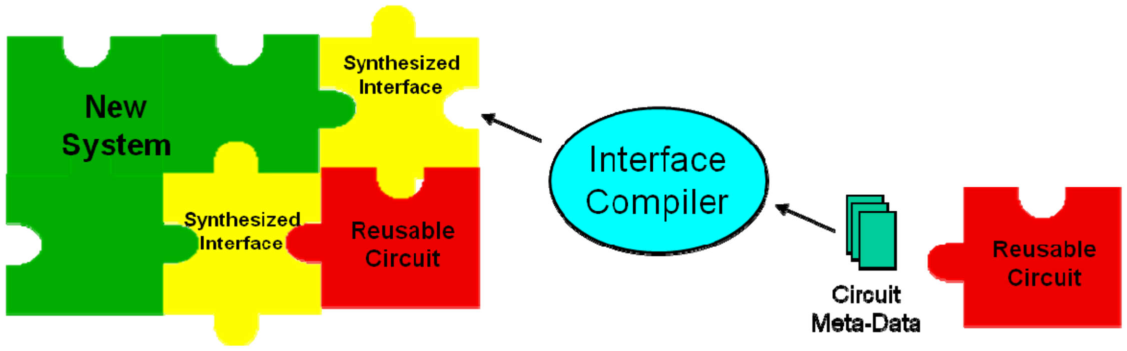

**Figure 10: An *Interface Compiler* Would Assume the Task of Creating the Logical Interface for a Reusable Component, and Integrate it into an Existing Design.**

Creating an interface compiler tool is a non-trivial task and would require solutions to a number of difficult issues. The following requisite issues must be addressed:

- Ability to formally characterize the interface of circuits in a machine readable form (i.e., a formal meta description),
- Creation of appropriate standards for describing the interface formally,
- Identification and characterization of a common set of interfaces,
- Development of synthesis and compilation techniques for reasoning about circuit interfaces and creating circuits to couple disparate interfaces, and

- The generation of libraries of cores with interface descriptions that adhere to the interface standard.

If solutions to these challenges are identified and techniques are created for automatically synthesizing circuit interfaces then the cost of reusing FPGA circuits will be significantly reduced. We expect that design productivity can increase by a factor of two if interface synthesis techniques are developed and reusable cores are made that exploit these standards.

## 4.2 Abstraction

Raising the level of abstraction means reducing the amount of detail that must be specified by the programmer. Since its inception, advances in computer science have proved that raising the level of abstraction leads to significant productivity gains. Programming for software systems has undergone a transition between many different levels of abstraction including machine code, assembly language, procedural programming languages, etc. Indeed, early gains of 5× in programmer productivity were reported as programmers moved away from assembly language toward PL/I and other higher-level languages. These productivity improvements came about for two reasons (24). First, the statements in higher-level languages are more powerful thereby allowing programmers to describe their application with fewer lines of code. Second, higher-level languages eliminate whole classes of bugs by automatically taking care of many low-level details. The bugs that remain are fewer in number and easier to find because they tend to be less obscure.

The productivity of digital circuit design has also increased significantly by exploiting higher level design abstractions. Digital circuit design has experienced a transition through several abstractions including design with individual transistors, design using logic gates within schematics, and register transfer level design using hardware description languages. A variety of new high level hardware design tools and methods are now emerging that build upon this trend (see Section 0 for a list of representative tools). These tools include high-level synthesis based on C or other procedural languages, graphical data flow design methods for DSP, and application-specific design compilers. Results from early adopters suggest that these tools do indeed improve design productivity if used appropriately.

While new abstractions are becoming available for digital design (i.e., the ASIC replacement use model), it is not clear that these abstractions will provide the revolutionary improvements in productivity needed for configurable computing. One reason for this is that many of these tools are essentially extensions of existing HDLs. They may remove some detail required with conventional VHDL or Verilog, but they still require an understanding of clocking, scheduling, pipelining, and other digital systems design concepts. Another reason is that these languages, while based on familiar programming languages such as C, have new concurrent semantics. A familiarity with the base language such as C may actually be a handicap when trying to learn these new semantics. Third, many of these abstractions are based on inherently sequential languages. The sequential nature of these languages limits the ability to specify and to exploit the massive parallelism available in hardware circuits (25).

Although these recent tools and languages are a step in the right direction, we believe that they are insufficient for moving hardware design to a significantly new level of design productivity. Additional advances in abstractions, languages, and compiler/synthesis tools are needed to increase productivity of FPGA based configurable systems. We propose several approaches that we believe may extend the advantages of abstractions. We believe that advances in these areas will provide over 2× improvement in design productivity.

### 4.2.1 Parallel Languages and Concurrent Models of Computation

It is well known that the incremental performance gains through architectural improvements of uni-processors is slowing and that microprocessors will not improve performance at the rate seen in the previous three decades (26). To address this trend, microprocessor manufacturers are using multiple processor cores within a single device to improve performance. Multi-core processors have the potential of achieving higher levels of performance with less power and cost. Multi-core processors, however, are more difficult to program than traditional uni-processors. Most programmers are taught to program using sequential languages and compilers struggle to exploit sufficient parallelism from such sequential descriptions. To address this issue, there is great interest in parallel programming languages and compiler tools for targeting multi-core architectures.

We believe that we have a unique opportunity to exploit this growing trend. We advocate the investigation and adoption of emerging concurrent programming approaches and models of computation for hardware design (27). The use of concurrent programming approaches will facilitate the extraction of the natural concurrency found within hardware circuits. Further, adopting standard concurrent languages will lead to more platform independent descriptions of algorithms that can be targeted to either hardware or parallel processor/multi-core systems.

While concurrent programming approaches are appropriate for both multi-core architectures and FPGA-based reconfigurable systems, the unique architectural features and constraints of FPGA-based systems may require unique concurrent programming approaches. To exploit the full advantage of the unique reconfigurable computing machine model may require custom concurrent programming constructs. Architectural issues that may impact the programming model include the distributed, non-uniform nature of the memory space, the availability of custom, non-standard functional units, and the ability to partially reconfigure the logic resources. Other research questions that should be addressed when investigating concurrent programming approaches for reconfigurable computing include the following:

- What unique concurrent programming structures are needed to support reconfigurable computing?
- Can emerging concurrent programming approaches be co-opted by reconfigurable computing or are fundamentally new concurrent programming approaches needed?
- How much of the underlying FPGA machine model needs to be exposed to the programmer?

We believe that FPGA design productivity can be increased by 2× by adopting concurrent programming approaches that facilitate design at higher levels of abstraction while preserving the underlying concurrency found within reconfigurable systems.

### 4.2.2 Multi-FPGA Synthesis and Compilation

Many configurable computing systems are designed with multiple-FPGAs to provide a large amount of computing performance. These systems integrate multiple FPGAs in a mesh, ring, systolic array or other topology to provide high levels of performance for computing problems that have a large amount of parallelism. While multi-FPGA systems provide a large amount of potential computing performance, they are more difficult to program than single FPGA systems. In addition to logic design, programmers of these multi-FPGA systems must manually partition the behavior between the various FPGAs in the system.

New high-level synthesis and compilation methods are needed to automatically target multi-FPGA systems. Most synthesis and compilation techniques assume a uniform array of logic and do not consider the impact of partitioning logic and computation between disparate FPGAs with limited connectivity. Future high-level synthesis approaches must consider the impact of inter-FPGA communication and perform coarse level partitioning and resource allocation based on the topology of the multi-FPGA system. Ideally, compilers for multi-FPGA systems would be able to target *any* multi-FPGA platform to facilitate the portability of configurable computing applications across different vendors and system topologies.

Figure 11 demonstrates how a multi-FPGA synthesis approach would work. The application-specific behavior is specified using the appropriate design language or abstraction. This behavior is specified with little or no platform specific annotations or descriptions (although a concurrent design language would be most effective). Before compilation, the programmer chooses a target platform which is described in an architecture description file (this file defines the FPGAs, memories, and other system resources). The compiler reads both the behavioral description and architecture description file to generate an executable on the target architecture. Unlike most traditional hardware compilers, this multi-FPGA compiler must perform logic and memory partitioning before the synthesis and technology mapping phases.

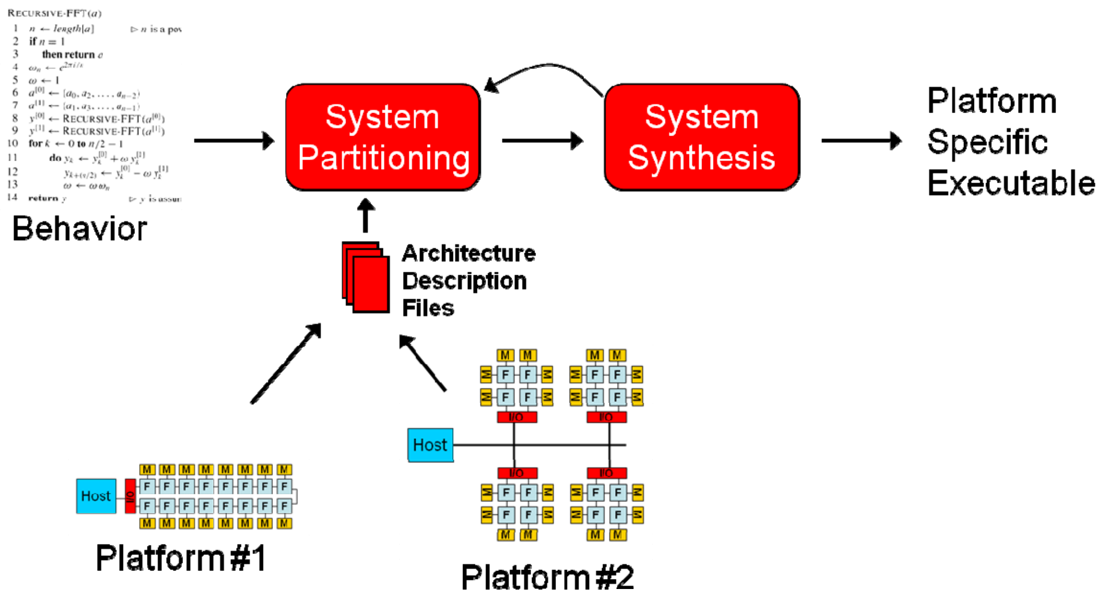

**Figure 11: Multi-FPGA Design Environment.**

Most multi-FPGA design environments require the user to perform FPGA partitioning manually. This manual partitioning step forces the programmer to make design decisions requiring a detailed understanding of the underlying FPGAs as well as good estimates of the size of the synthesized hardware. We believe that with advances in behavioral synthesis and partitioning techniques, much of this partitioning can be automated to simplify the design process and substantially increase design productivity.

## 4.3 Turns Per Day

There is a big difference between debug productivity for software and debug productivity for hardware. In a typical FPGA hardware design flow, we achieve one to two debug iterations in a given day. With a software development tool such as *gcc,* it is possible to achieve more than 20 debug iterations per day. In fact the number 20 was chosen somewhat arbitrarily and likely is much higher, especially if one counts the use of *printf()*-based runs as debug iterations.

One of the key issues with regards to hardware debug is that there are actually two development cycles that the designer must navigate (see Figure 12). On the left is a debug cycle that approximates software development, consisting of compile, simulate, modify design, and repeat. Once this has been done to the designer's satisfaction he/she moves to the cycle on the right which consists of synthesis/place-and-route/timing-closure/download followed by hardware execution and often confusion. These are two very different types of debug cycles. The simulation cycle on the left is very slow to simulate but provides excellent visibility into the operation of the circuit. The cycle on the right runs thousands of times faster but provides very little visibility into the operation of the circuit.

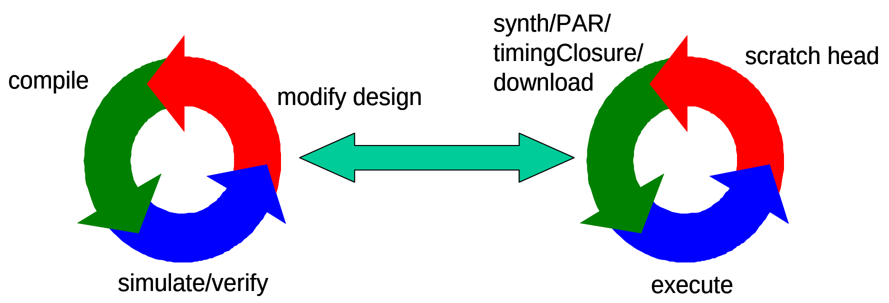

**Figure 12: Configurable Computing Development Cycle.**

One of the chief difficulties with this hardware design cycle is the difficulty of conducting *what-if* experiments. Such experiments are an important part of many design processes, and are exceedingly difficult in hardware design. To perform such an experiment, the user modifies his/her design code, and then may spend significant amounts of simulation time to determine whether the experiment will be successful. Often however, he/she must do the experiment in hardware which requires even more additional time to synthesize and implement the circuit before the experiment can even be run. In either case running such an experiment may take multiple hours. In short, most hardware design environments do not encourage interactive development.

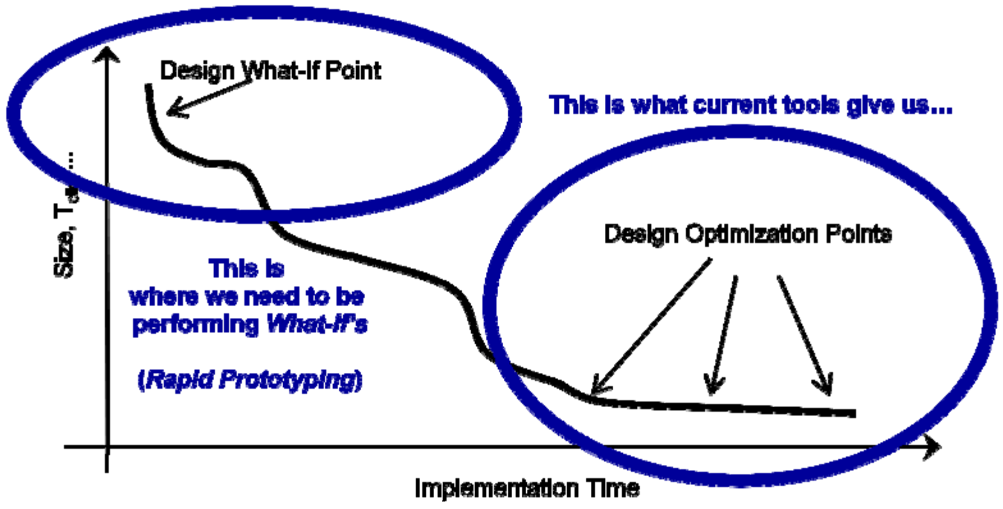

**Figure 13: CAD Tools and Design "What-If Experiments".**

The chief reason for this is that current CAD tools simply do not support interactive development. As shown in Figure 13, current CAD tools have been developed to produce designs on the extreme right side of the implementation time/circuit area space. That is, they focus on providing the smallest implementation but at the cost of long run times. While appropriate for final implementations, this does not support the idea of rapid prototyping or what-if experiments.

A second difficulty with hardware development environments is a lack of infrastructure. As shown on the right side of Figure 14, typical software development environments have mature tools available for use, with many choices available. In contrast, hardware development environments are missing groups of tools. In addition, the tool choice on the hardware side is often very limited and the tools themselves not of high quality.

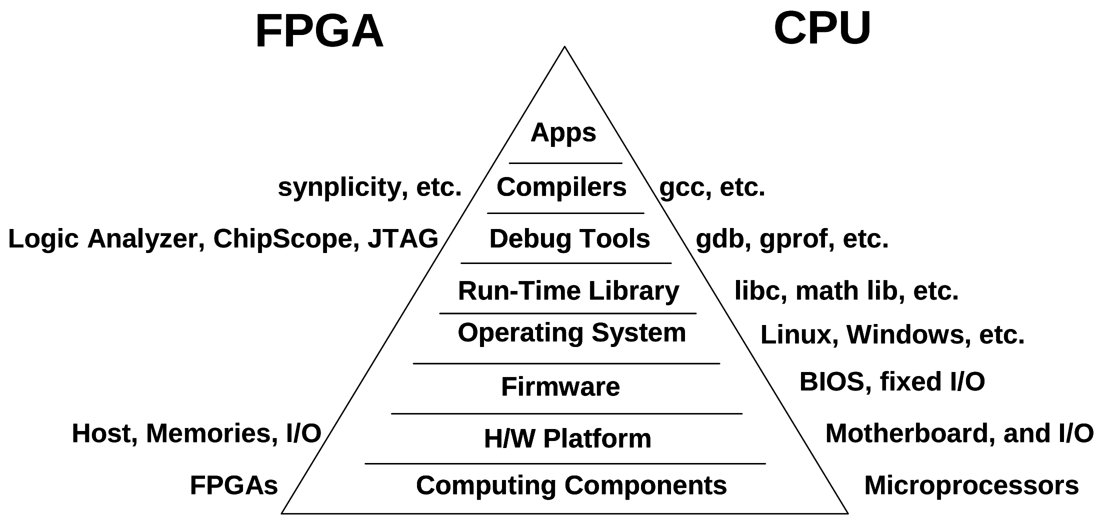

**Figure 14: Sparse Infrastructure for Configurable Computing Systems.**

It is our belief that the impact of improved debug infrastructure for increasing the number of debug turns per day cannot be overstated. If we could increase the number of turns per day by 3 times, one could say that we would experience a 3 times increase in design productivity. However, the effect may be much greater. Increasing the number of turns per day in the debug environment has a systemic effect on the entire design process. Users no longer are forced to multitask while waiting for long implementation runs to complete. Rather, they can focus on the debug task, rapidly iterating with *what-if* scenarios and experiments and greatly multiplying their current capabilities. Thus, we believe that improving debug infrastructure may provide a nonlinear impact and give a much greater than 3 times productivity improvement, and mitigates the unproductive "busy-wait" mode of development characteristic of contemporary practices. Below we provide a number of approaches which we believe should be investigated to increase the number of turns per day a hardware designer can achieve.

### 4.3.1 Standard Platform Services

In comparing standard computing platforms with configurable computing platforms we see that huge differences exist in the support provided between the two. Computer systems provide extensive services to the user, often without the users being aware of this. These services are provided by a combination of hardware support, firmware support, and software support. These include things such as device interface capabilities (device drivers), networking stacks, timers and interrupt capabilities, self check and monitoring capabilities, run levels, linkers and loaders, and debug support. In contrast, configurable computing support for such services is severely limited. Some platforms provide few, if any, of these services; even when some support is available is nonstandard between platforms, and the availability of such services is uneven. As a result, users cannot depend upon a "standard" set of services.

This lack of services comes with a large opportunity cost. Since every platform is a custom platform, there is no third-party software development industry being built up for configurable computing similar to what is available for conventional computing. In general, the users are at the mercy of individual board vendors to such capabilities. As previously shown in Figure 14, the result is very sparse support.

Support for standard system services would greatly change how a user used a configurable computing platform. As shown in Figure 15, in the creation of the user's application he would specify the services required either explicitly or implicitly. These services could include I/O interfaces, memory interfaces, timers, interrupts, etc. The compiler would determine what services were required and integrate the appropriate intellectual property to create those services in hardware, linking them to the user's design as needed. Importantly, the compiler would automatically create the interfaces. As a result, user designs would merely specify services required and those would be automatically integrated to the user design, similar to how software libraries are linked in with minimal effort on the part of the user.

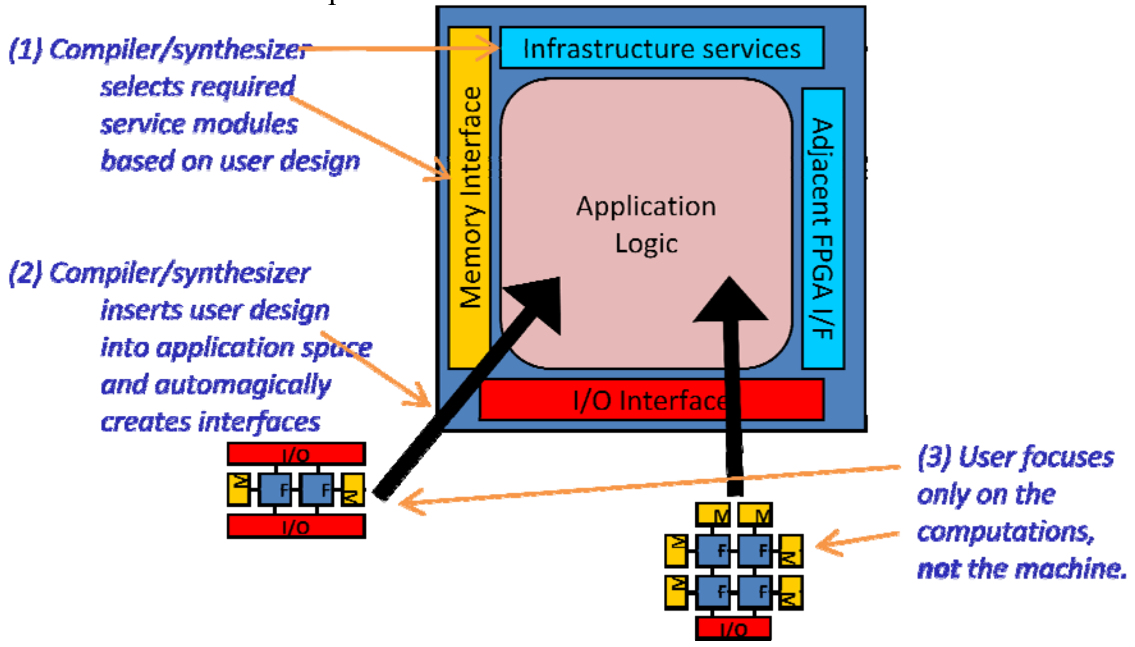
(1) Compiler/synthesizer selects required service modules based on user design
Infrastructure services
Memory Interface
Application Logic
Adjacent FPGA I/F
(2) Compiler/synthesizer inserts user design into application space and automagically creates interfaces
I/O Interface
I/O
F
F
M
M
I/O
(3) User focuses only on the computations, not the machine.
M
M
F
F
I/O

**Figure 15: Standard System Services Support.**

Debug is so important that we believe it provides its own set of requirements. For example, the JHDL system provides an example of hardware-in-the-loop debug capabilities which greatly simplifies configurable computing debug (28). By providing a simulation/runtime API, it allows the same suite of tools to be used to debug a design either in simulation or in hardware execution (see Figure 16). When simulating, all computation of next state values is done by the built-in JHDL simulator and the simulation infrastructure used to display circuit state in various GUI windows. In hardware mode, however, commands to advance execution cause commands to be sent to the hardware platform (onto which a bitstream was previously configured). The state values from the executing circuit were then retrieved from the hardware platform using *readback*. The state values received through readback are back-annotated into the simulator data structures for display. This provides a standard platform around which to create debug tools and other aides, which operate in both hardware and simulation modes.

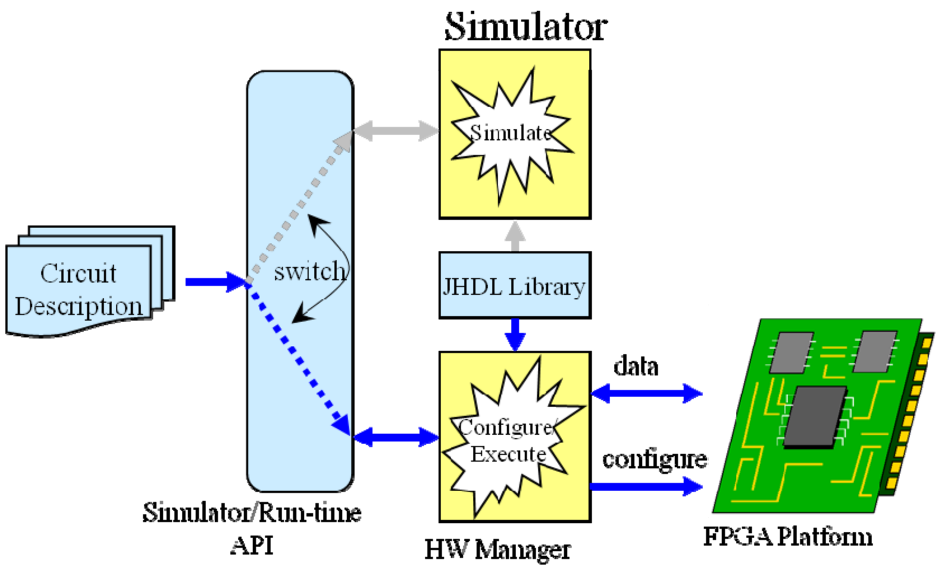

**Figure 16: Hardware-in-the-Loop Hardware Debug.**

This entire facility is based on the creation of an intermediate circuit data structure which can be used for both simulation and hardware execution. This provides a standard data structure to which user-created tools can be interfaced. This is in contrast to today's CAD tools where intermediate formats are fiercely protected by vendors as proprietary data, providing no possibility for third-party software development to be done to aid in the debug process.

Given that such an intermediate format and tool infrastructure exists, however, it becomes straightforward to create very powerful runtime facilities to provide the system services described above. For example, Figure 17 illustrates the use of checkpointing a computation. Checkpointing relies on the ability to extract the complete state of a running computation and later restore it, something that was demonstrated in JHDL.

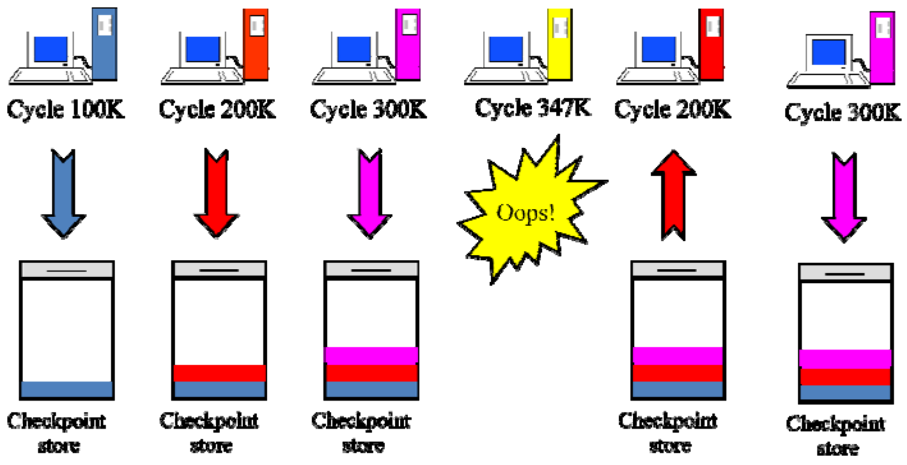

**Figure 17: Checkpointing of Hardware Computations.**

Finally, such capabilities can be leveraged to support what-if experiments in debug where on-the-fly creation of debug circuitry via bitstream manipulation is used to provide the user with unprecedented access to the internal state of a running computation.

A major problem with today's CAD tools is that they make little provision for debug, typically obfuscating their operation and intermediate file formats, and thereby preventing users from adding such debugging aids on after the fact. Importantly, we believe support for debugging at runtime such as we have outlined above will not come for free — a few percentage increase in circuit area should be a good trade off for large gains in design productivity, something the software world accepted years ago. We believe that effective debug and run time support infrastructure can be created for configurable systems but this infrastructure can only succeed if it is built into the design process and CAD tools *from the outset*.

### 4.3.2 Firmware

We propose the use of RC "firmware" to significantly simplify the design and debug process. This is illustrated in Figure 18, where the I/O interfaces around the periphery of a chip are standardized. These circuits can even be precompiled onto the chip itself and may be application-independent. User designs are then compiled and, using partial configuration or design merging, are configured onto the chip and wired up to the standardized interfaces. The benefits of such an approach would be much faster place-and-route, the possibility of the creation of a platform-independent design flow, enhanced portability, and increased reuse. We understand that such approaches have been tried by vendors in the past, and it is our belief that these have failed because they may have included too much circuitry and thus impacted the ability of a designer to place a significant design in the remaining circuit area. The approach we propose would rely heavily on synthesis and CAD tools to only insert the standardized I/O interfaces which were required for a given design, leading to maximum circuit area available for user designs.

This approach is closely related to the notion of incremental design. Stated another way, supporting firmware requires the same CAD tool support that supporting incremental design requires. That is, the CAD tool flow needs to support pre-existing placed and routed circuitry which can be left intact while additional circuitry is synthesized and placed and routed around it. The notion of firmware could then be extended to the idea of performing partial re-place and re-routing of an existing design. An important observation is that this is currently prohibited by the typical CAD tool flows found in commercial tools, which flatten the entire design heart hierarchy as the first step in the synthesis process. We believe that by preserving the design hierarchy through the entire tool flow it will be possible to create designs which have locality of placement which matches the design hierarchy better, allowing localized changes to the design source to be reflected in minimal amounts of replacement and rerouting of the circuit – the foundation of an incremental design flow.

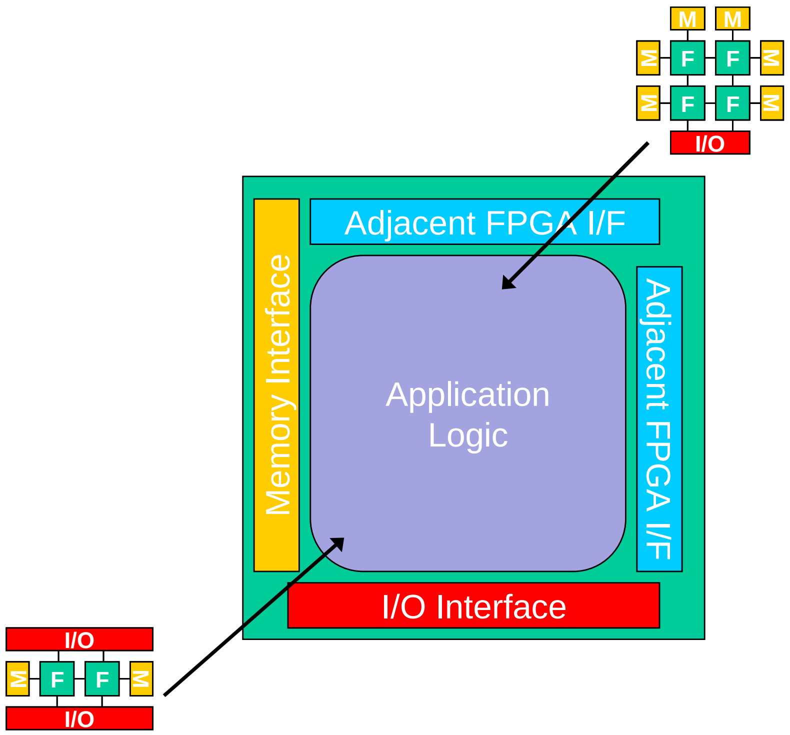

**Figure 18: RC Firmware.**

### 4.3.3 High-Level Abstraction Debug

When debugging a configurable computing application, there are two choices given to a user. The first is to use a “simulator” which executes at a small fraction of the target operating frequency of the final application. A simulation-based debugging environment, however, provides essentially perfect visibility of the design and perfect controllability over the executing application. The user is allowed to use file input and output as well as other general computing aids to help in the creation of input stimulus and the analysis of output results. In addition the user is able to change variables to perform what-if scenarios, etc. The alternative to simulation is to “execute” the circuit at the application speed. The obvious benefit of this is the speed of execution – the user can boot operating systems on the platform, or run the app in its entirety in relatively short amounts of time. The disadvantage of this approach is that the user has little control of the execution and limited visibility of the circuit. New methods and techniques are needed to provide the visibility and controllability of a simulator to the run-time environment of an actual system.

The key problem preventing this is the lack of information shared through the entire implementation toolchain (see Figure 19). In this figure, vendor of compiler “X” has its own internal file formats and database to store the information related to the front-end compilation step. However a second vendor (vendor “Y” in the figure) provides the synthesis tool with its corresponding proprietary file formats and database. Finally, FPGA vendor “Z” provides the implementation tools and its corresponding file formats. These file formats and databases are largely undocumented, proprietary, and unavailable to the end-user. As a result, it can be very difficult to relate values found in a readback bitstream (from vendor “Z”) to the original design source (from vendor “X”).

**Figure 19: Multiple Design Databases in Typical FPGA Design Flow.**

The approach we propose here, called "high-level abstraction debug" is to provide a unified set of files, databases, and APIs for the entire design flow. With such a unified database, the translation steps from source code to bitstream can be documented and used by the creator of debug tools to provide information linking bitstream contents to original divine source. This unified database is shown in Figure 20. These debug tools will allow the user to debug at the original source code level and provide debug which match the models of computation embodied in the original high-level abstract design source.

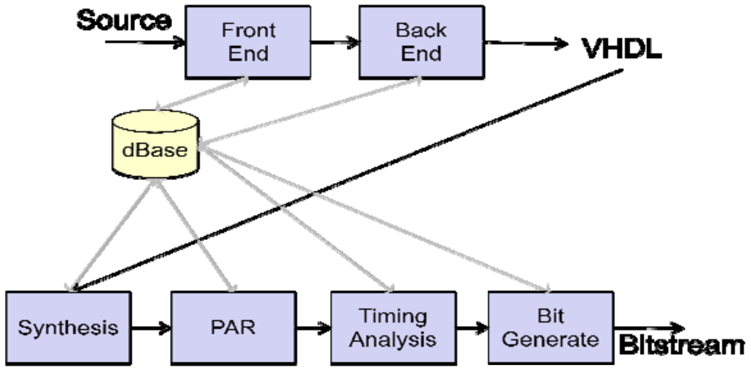

**Figure 20: Unified Database for Cross Tool Linking.**

In summary, debug and runtime aids can only be successful if they are built into the design process and CAD tools from the outset. A major problem with today's CAD tools is that they make little provision for debug, typically obfuscating their operation and intermediate file formats, and thereby preventing users from adding such debugging aids on after the fact. Importantly, we believe support for debugging runtime such as we have outlined above will not come for free — a modest increase in circuit area should be a good trade off for large gains in design productivity, something the software world accepted years ago.

### 4.3.4 Summary of Research Approaches

The approaches described in the previously section define the research areas we feel are most important to address in order to substantially increase the design productivity of FPGA-based systems for configurable computing machines. Each of these areas is interconnected as shown in Figure 21 and design productivity will significantly increase *only* if advances are made in each of these areas and applied at all levels of the design methodology. We believe that advances in each of these areas will provide up to a 25× improvement in design productivity.

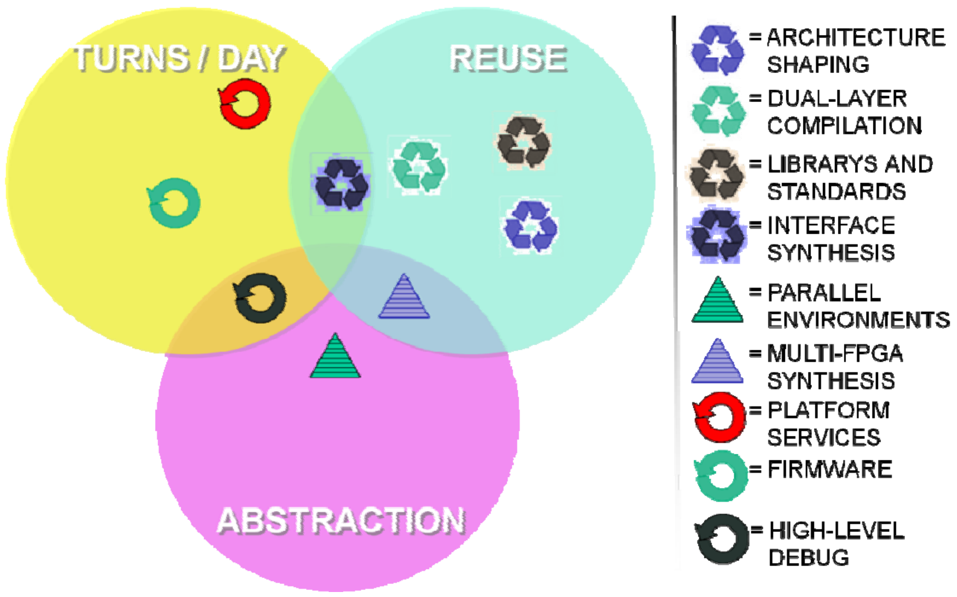

**Figure 21: Relationship between Research Approaches.**

# 5 Integrated Research Vision

During the course of this effort, two study teams[3] have been funded by DARPA, each charged with defining a vision and roadmap for addressing fundamental challenges in application development tools for FPGA-based systems. The outcomes from these two studies are described in two reports entitled *Strategic Infrastructure for Reconfigurable Computing Applications (SIRCA)* and *FPGA Design Productivity (FDP)*. The purpose of this section is to describe an integrated research vision that includes the major concepts and research approaches from these two studies in a unified and integrated manner.

The two study teams met together on June 5th, 2008 in Salt Lake City along with experts in the field to present the results of their findings and begin the task of integrating the research vision presented by both teams. Breakout groups at the meeting provided feedback and suggestions on how to integrate the results from these research studies. We believe that this unified vision forms the basis of a research vision that will lead to revolutionary improvements in design productivity for reconfigurable computing systems.

The two teams worked independently to query the reconfigurable computing community, gain a solid understanding of contemporary practices, and research past and current endeavors related to FPGA design productivity. Surprisingly, the two teams presented findings that shared several common themes. Both teams discussed similar causes to the problem and presented similar approaches for addressing the challenges in application development for FPGA-based systems. However, each team approached its study in a unique manner and emphasized different aspects of the design methodology. While the emphasis of each study was different, the results of both studies complement each other well and when taken together present a clear and complete research plan for significantly improving FPGA design productivity.

The *SIRCA* team organized its study around the concepts of Formulation, Design, Translation, and Execution (FDTE). This research model is defined horizontally in terms of the four fundamental stages in application development. The SIRCA study emphasizes research challenges in all four of these development stages but especially the Formulation stage, which features strategic design exploration and tradeoff analyses for complex systems and is pivotal for design productivity in many fields of engineering, and yet routinely overlooked in conventional hardware and software engineering.

The *FDP* team organized its study around three research focus areas: Abstraction, Reuse, and Turns per day (ART). This research model is defined vertically, where each research focus area defines a key research thrust that must be addressed in *all* stages of application development. The FDP study emphasizes the need to increase abstraction (reduce design detail), apply reuse, and reduce turns per day at all stages of the design process to obtain significant improvements in design productivity.

Figure 22 visually demonstrates the relationship between the models presented by the two study teams. In the center, application development is defined in terms of the four stages in the FDTE model. The process begins with Formulation, featuring strategic exploration of candidate algorithms and architectures supported by performance

[3] The two teams funded by DARPA include a team from Brigham Young University and Virginia Tech and a team from University of Florida, George Washington University, and Clemson University.

prediction for tradeoff analyses. After strategic decisions are made, the process moves to code design and implementation in the Design stage, then Translation to produce an executable form, and finally Execution, where verification and optimization occur and the application executes supported by a variety of run-time services. The arrows between stages emphasize the iterative nature of the development process and importance of exploiting results (templates, libraries, patterns, run-time information, etc.) between stages.

Each of the three research themes of the ART model are shown as vertical bars that span all development stages of the FDTE model. Reuse, for example, can be applied during Formulation, Design, Translation, and Execution to significantly reduce the amount of new work that must be performed by a programmer or by automated design tools. The other two focus areas, abstraction and turns per day, also span the four design stages of the FDTE model – technical approaches for each of these focus areas are possible at each design stage to improve programmer productivity.

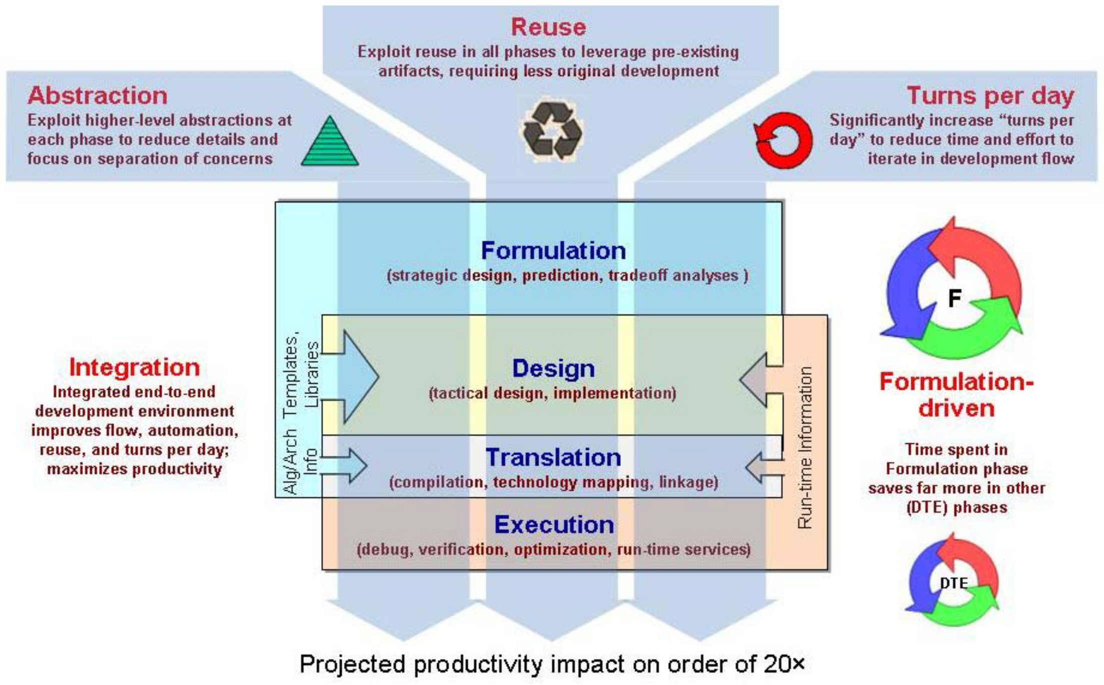
Reuse
Exploit reuse in all phases to leverage pre-existing artifacts, requiring less original development
Abstraction
Exploit higher-level abstractions at each phase to reduce details and focus on separation of concerns
Turns per day
Significantly increase "turns per day" to reduce time and effort to iterate in development flow
Formulation
(strategic design, prediction, tradeoff analyses )
Design
(tactical design, implementation)
Translation
(compilation, technology mapping, linkage)
Execution
(debug, verification, optimization, run-time services)
Alg/Arch Templates, Libraries
Info
Run-time Information
Integration
Integrated end-to-end development environment improves flow, automation, reuse, and turns per day; maximizes productivity
F
Formulation-driven
Time spent in Formulation phase saves far more in other (DTE) phases
DTE
Projected productivity impact on order of 20×

**Figure 22: Integrated Research Vision.**

Each of the teams identified a set of specific research thrusts that will lead to major improvements in design productivity. Taken together, 21 research thrusts were identified. As highlighted in Table 1, each of these research thrusts can be placed within the integrated research vision of Figure 22. The two study teams believe that improvements in design productivity of 20× or better are possible if advancements are made with each of the development stages of the FDTE model and focused in terms of abstraction, reuse, and turns per day.

| Thrusts | ART Model | | | FDTE Model | | | |
|---|---|---|---|---|---|---|---|
| | Abstraction | Reuse | Turns/day | Formulation | Design | Translation | Execution |
| ***SIRCA Research Thrusts*** | | | | | | | |
| 1. Strategic exploration | X | X | X | X | | | |
| 2. High-level prediction | X | | X | X | | | |
| 3. Numerical analysis | X | X | | X | | | |
| 4. Bridging design automation | | X | X | X | X | | |
| 5. System-level parallel languages | X | X | | | X | X | |
| 6. HW/SW codesign methods | X | X | | | X | X | |
| 7. Reusable & portable design | | X | | | X | X | |
| 8. Translation algorithms | | | X | | | X | |
| 9. Translation target architectures | | | X | | | X | |
| 10. Runtime debug & verification | X | | X | | X | | X |
| 11. Performance analysis | X | | X | | X | | X |
| 12. Run-time services | X | | X | | | | X |
| ***FDP Research Thrusts*** | | | | | | | |
| 1. Architecture shaping | | X | | | | X | X |
| 2. Dual-layer compilation | | X | | X | X | | |
| 3. Libraries & standards | | X | | | X | X | |
| 4. Interface synthesis | | X | X | | X | X | |
| 5. Parallel environments | X | X | | X | X | | |
| 6. Multi-FPGA synthesis | X | X | | X | X | | |
| 7. Platform services | | | X | | | | X |
| 8. Firmware | | | X | | | X | X |
| 9. High-level debug | X | | X | | | | X |

**Table 2: Research Thrusts and Models**

# 6 References

1. **DOD Advisory Group on Electronic Devices.** Special Technology Area Review on Field Programmable Gate Arrays (FPGAs). July 2005. ARL-SR-147.
2. **Alan Allan, Don Edenfeld, William H. Joyner, Andrew B. Kahng, Mike Rodgers, and Yervant Zorian.** 2001 Technology Roadmap for Semiconductors. *IEEE Computer.* 2002. Vol. 35, 1, pp. 45-53.
3. **DeHon, Scott Hauck and Andre.** *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing.* s.l. : Morgan Kauffman, 2007. Chapter 21.
4. *System Level Tools for DSP in FPGAs.* **James Hwang, Brent Milne, Nabeel Shirazi and Jeffrey D. Stroomer.** s.l. : Springer Berlin, 2001. Field Programmable Logic and Applications (FPL). pp. 534-543.
5. **David Andrews, Douglas Niehaus, and Peter Ashenden.** Programming Models for Hybrid CPU/FPGA Chips. *IEEE Computer.* January 2004, pp. 118-120.
6. *Mapping a Domain Specific Language to a Platform FPGA* . **Chidamber Kulkarni, Gordon Brebner, and Graham Schelle.** s.l. : ACM, 2004. Design Automation Conference. pp. 924-927.
7. *Attig, M. Dharmapurikar, S. Lockwood, J.* . **Attig, M. Dharmapurikar, S. Lockwood, J.** s.l. : IEEE, 2004. Field-Programmable Custom Computing Machines (FCCM). pp. 322- 323.
8. *Using General-Purpose Programming Languages for FPGA design.* **Nelson, B. Hutchings and B.** 2000. Proceedings of the 37th Design Automation Conference (DAC). pp. 561-566.
9. **Nierstrasz, S. Moser and O.** The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer.* September 1996, Vol. 29, 9, pp. 45-51.
10. **Boehm, B. W.** Managing Software Productivity and Reuse. *IEEE Computer.* September 1999, Vol. 32, 9, pp. 111-113.
11. *A general economics model of software reuse.* **Cruickshank, J. E. Gaffney and R. D.** Melbourne, Australia : ACM, 1992. Proceedings of the 14th International Conference on Software Engineering. pp. 327-337.
12. *Enabling reuse-based software development of large-scale systems.* **Selby, R. W.** 6, June 2005, IEEE Transactions on Software Engineering, Vol. 31, pp. 495-510.
13. *The impact of tools on software productivity.* **T. Bruckhaus, N.H. Madhavii, I. Janssen, and J. Henshaw.** 5, s.l. : IEEE, September 1996, IEEE Software, Vol. 13, pp. 29-38.
14. *Design Productivity for Configurable Computing.* **B. Nelson, M. Wirthlin, B. Hutchings, P. Athanas, and S. Bohner.** July 2008, Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA).
15. **Poulin, Jeffrey S.** *Measuing Software Reuse: Principles, Practices, and Economic Models.* s.l. : Addison Wesley, 1997.
16. **Ivar Jacobson, Martin Griss, and Patrik Jonsson.** *Software REuse: ARchitecture, Process and Organization for Business Success.* s.l. : ACM Press, 1997.
17. **Tracz, Will.** *Confessions of a Used Program Salesman: Institutionalizing Software Reuse.* s.l. : Addison Wesley, 1995.

18. *Increasing design quality and engineering productivity through design reuse.* **Carlson, Emil Girczyc and Steve.** 1993. Proceedings of teh ACM IEEE Design Automation Conference (DAC).
19. *HPC Productivity: An Overarching View.* **Kepner, Jeremy.** 4, 2004, International Journal of High Perforamnce Computing Applications, Vol. 18, pp. 393-397.
20. *Design patterns for reconfigurable computing.* **Andre DeHon, Josua Adams, Micael DeLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton.** s.l. : IEEE, 2004. IEEE Symposium on Field Programmable Custom Computing Machines (FCCM). pp. 13-23.
21. *OpenFPGA corelib core library interoperability effort.* **M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Prichard, P. Palazzari, and G. Kuzmanov.** 2007. Proceedings of the 2007 Reconfigurable Systems Summer Institute (RSSI).
22. *VSIPL: An Object-Based Open Standard API for Vector, Signal, and Image Processing.* **R. Janka, R. Judd, J. Lebak, M. Richards, and D. Campbell.** 2001, IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 2, pp. 949-952.
23. *The Morphware Stable Interface: A Software Framework for Polymorphous Computing Architectures.* **D. Campbell, D. Cottel, R. Judd, K. MacKenzie, and M. Richards.** 2003. Seventh Annual Workshop on High Performance Embedded Computing.
24. **Brooks, Fredrick P.** *The Mythical Man-Month: Essays on Software Engineering.* s.l. : Addison Wesley, 1995.
25. **Edwards, Stephen A.** The Challenges of synthesizing hardware from C-like languages. *IEEE Design & Test of Computers.* 2006, pp. 375-386.
26. **K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelik.** *The Landscape of Parallel Computing Research: A View from Berkeley.* EECS Department, University of California at Berkeley. 2006. UCB/EECS-2006-183.
27. *A Framework for Comparing Models of Computation.* **Sangiovanni-Vincentelli, Edward A. Lee and Alberto.** 12, December 1998, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, pp. 1217-1229.
28. *Designing and Debugging Custom Computing Applications.* **Brad Hutchings, Brent Nelson, and Mike Wirthlin.** 1, s.l. : IEEE, January-March 2000, IEEE Design and Test of Computers, Vol. 17, pp. 20-28.
29. **Black River Systems Company.** FPGA Tool Taxonomy Final Report. June 2007.
30. *High-Quality, Deterministic Parallel PPlacement for FPGAs on nCommodity Hardware.* **Adrian nLudwin, Vaughn Betz, and Ketan Padalia.** s.l. : ACM, 2008. ACM International Symposium on Field-Programmable Gate Arrays. pp. 14-23.

# Appendix

## A.1 Survey of Hardware Metrics

This appendix provides a sampling of papers identified in the literature which illustrate the state of the art in hardware design metrics and descriptions of those papers. The discussion for each paper is not meant to be a summary but rather identifies relevant points for FPGA productivity metrics. The papers are listed in alphabetical order of the primary author.

*The High-End Computing Productivity Crisis*, The High-End Crusader (anonymous), HPCWire, April 16, 2004.

This article notes the following trends in HPC: each new parallel machine requires ever increasing levels of programming skill to successfully program, increasing time is required to program each new machine, the number of users of the highest-end machines is decreasing and the number of programs written for those machines is decreasing. The author argues that architecture, combined with programming language is the key to increasing productivity. He describes Type-T algorithms (limited communication and well-balanced workloads) and Type-C algorithms (long-range communication and poorly-balanced workloads). He also describes Type-T architectures (weakly-parallel processors and low-BW system interconnect) and Type-C architectures (strongly-parallel processors and high-BW system interconnect). He then makes the obvious point that running a Type-C job on a Type-T system is a bad idea. The point is that productivity can be enhanced if architecture is included. For metrics, he includes three times: programming time, execution time, and results analysis time.

*FPGA Tool Taxonomy Final Report*, Black River Systems .

This report suggests a large number of "Measures of Performance" (MOPS) for evaluating tools. The report is not proposing "productivity" metrics, rather it is proposing questions that should be asked about a given tool. It is interesting to note that they do not call them metrics. All of these "MOPs" are of the form, "Does the tool <do such and such>?". One example, "Does the tool support parallel computing?" The report lists many "measures" for evaluating tools (p. 20-26). Many of these are questions to ask when evaluating a tool and do not lend them selves to quantitative measurement. Interesting points they make in the report include: 30%-40% of design time is HDL coding, 60%-70% of the time is spent in verification.

Graphical tools allow users to more easily reuse IP and achieve a quick and dirty solution more quickly (page 14). High-level languages offer productivity because fewer lines of code are needed to code the behavior. In addition, there is less detail to manage with HLL than HDL (page 15), easier to port/maintain (page 16) Abstraction enhances portability (page 16) HLL's limit the designer's ability to "tweak" the implementation for improved area/speed. Abstracting hardware issues limits ability to take advantage of hardware-specific features (p. 16). A number of these outcomes are not unexpected and point out the difficulty of reducing design time without reducing circuit performance.

*Software Metrics Lead the Way to Better HDL Coding Practices*, Gregory P. Chapelle and Michael L. Lewis, EETimes, 1999.

This editorial argues for the use of software metrics as a way of increasing design productivity. However, the method proposed looks to be more of a management tool to help understand the current progress and phase of the development effort. It proposes the automated collection of design statistics regularly as the design progresses (lines of code, lines of comments, white space lines, …). They argue that by monitoring the time evolution of these statistics, management can learn whether the design is continuing to progress in a predictable, healthy way. To the extent that this careful monitoring can enhance productivity by avoiding common pitfalls or by allowing management to add more resources when it becomes obvious such resources are needed, this approach has merit. In many ways, it is similar to the METRICS paper. However, it does not propose any new method for coding which will increase productivity, rather it simply argues for close monitoring of progress. Such monitoring will prevent surprises and make the overall process more predictable.

*Comparative Analysis of High Level Programming for Reconfigurable Computers: Methodology and Empirical Study*, El-Araby et al, 3rd Southern Conference on Programmable Logic (SPL), 2007, pp. 99-106.

This paper from the GWU CHREC Group attempts to balance the trade-off between design quality and design productivity. They have an "ease of use metric" based on total acquisition time (time to learn tool and gain experience) and total development time. They have seven equations they use to get this "ease of use" value. They obtain data by having different students with various levels of experience creating designs with the tool. It represents a useful and interesting example of a tool evaluation process.

Ideas gleaned from the above papers include the following:

- The line between metrics and productivity discussions is often blurred. Some focus on raw metrics while others talk about metrics only in the context of their productivity approaches.
- Skill levels not uniform across all stages of development process.
- Core reuse, while an important problem to solve, is a difficult problem to solve for both technical and non-technical reasons.
- Utility/cost was a new metric for us. It allows for a variety of ways of describing productivity.
- While most papers focus on LOC, there are a few out there which argue against it.
- Time-to-first-solution is an interesting metric that measures how quickly one can get up and experimenting with an implementation.
- There are different types of workflows (research-oriented to production-oriented)and each has different needs w.r.t. productivity tools.

*The METRICS System*, Fenstermaker et al, DAC 2000.

This paper describes METRICS, the authors' system to support continuous design process optimization (DPO). METRICS gathers data about the design as it progresses and stores it in a way to support analyses and predictions of success for the project. It points out that CAD tools do not typically provide the data required to support DPO, and, those tools that provide some of the data required all provide it in different formats. They also argue that at the current time we do not know what data should be collected. In the end, they argue that standard tool metrics will be required if DPO is to become commonplace as it is in other areas (semiconductor manufacturing, for example).

*A Relative Development Time Productivity Metric for HPC Systems*, Funk, Kepner, Basili, Hochstein, Lincoln Laboratory and Univ of Maryland.

This was a paper on HPC Productivity Metrics at the HPEC 05 conference. They propose 4 axes of productivity:

1) Performance of the final implementation,
2) Programmability (time from idea-to-first-solution),
3) Portability (transparency) of the solution,
4) Robustness (reliability).

Point 2) above has been proposed by others. It places an emphasis on getting to an initial solution quickly. They propose a productivity formula on slide 6 that is: PRODUCTIVITY = UTILITY/COST. In this formula, UTILITY is the value a user places on getting a result at time T. UTILITY is thus a time-varying function that reflects that solutions arrived at different times have different value to the user. The COST term includes machine cost + operating costs + software development cost.
They point out that for small codes, productivity is simply the final application performance divided by the cost of writing the code. The results they show include such statements as these: "… OpenMP is more productive than other approaches for small numbers of CPUs in a shared memory architecture.", "… for larger systems MPI and Co-Array Fortran (CAF) scale well", and "Performance of C+MPI and pMatlab is comparable".

*Measuring Productivity and Quality in Model-Based Design*, Arvind Hosagrahara and Paul Smith, The Mathworks.

This paper is from MathWorks and advertises their approach for measuring productivity and quality in their control system design tools. They emphasize that LOC is the basis for all SW productivity measures such as LOC/unit and defects/LOC, but that such a metric may be misleading with model-based design. For our study, the issue here is that in Model Based Design (MBD), a model is manipulated and then code is automatically generated from that model. LOC measurements on this automatically generated code are not as useful as when all code was hand-written. Thus, new metrics are needed when high level programs emit code. He argues that the new metrics should all focus on time spent and defects introduced rather than measurements of the size of the code. The same metrics can also apply to later design changes (how hard were they to make, did they introduce new defects?). This was the first software paper we found that didn't focus on LOC in the end.

*Emerging DoD Sensor Processing Requirements*, Jeremy Kepner, April 2006.

This is a presentation on Emerging DoD sensor requirements and the need for higher throughput processing and was presented at the DARPA Workshop in April 2006. It summarizes the various DARPA programs and their benchmark suites (HPC Challenge, HPEC Challenge, Compact Apps, …). With regard to FPGA design, it points out that the skill level required for each step of the FPGA development process is different (PhD vs. MS vs. BS) and that currently, the FPGA development process is not portable. Finally, it shows that the FPGA development process typically consists of three different four-month phases.

*HPCS Application Analysis and Assessment*, Jeremy Kepner and David Koester.

This work was a presentation to DARPA on HPCS, and examines a number of useful points about software productivity metrics. They define productivity as the ratio of utility to cost. The main metric discussed is lines of code (LOC). One plot shows the LOC required for various implementation options (OpenMP, MPI, etc) for a set of benchmarks. Another shows the achieved performance for NAS FT as a function of lines of code – the message is that more complex programming tasks (a FORTRAN implementation on 16 CPU's using MPI for example) get correspondingly higher performance than simpler programming tasks (a Java implementation on 16 CPUs or a uniprocessor implementation for example).

An interesting metric proposed is a $\Delta x$ vs. $\Delta y$ metric where $\Delta x$ is the code change and $\Delta y$ is the benefit achieved. The paper also suggests measuring distance between code changes to determine if changes are localized (good) or distributed (bad). Slide 24 then lists a collection of productivity models from the software community. This is a good jumping-off point to look at a variety of metrics. Slide 25 indicates that code size is the most important SW productivity parameter, and that HPC can reduce code size in two ways: by using higher level languages and by reuse. Similar to the need for performance in FPGA-based systems, the paper indicates that HPC performance requirements limit the exploitation of these two ideas.

Measures of success proposed include: (a) that the results are accepted by users, vendors, … and (b) that they can quantitatively explain HPC rules of thumb such as: "OpenMP is easier than MPI but doesn't scale as well". Much of these slides directly mirrors much of what is being talked about with respect to FPGA design productivity.

*HPC Productivity: An Overarching View*, Jeremy Kepner, International Journal of High Performance Computing Applications, Vol. 18, No. 4, 393-397 (2004).

In this preface, Kepner argues that there are 3 different kinds of workflows that must be considered. In the researcher workflow the focus is on knowledge discovery with rapid design iterations, similar to that founding rapid prototyping. In the enterprise workflow, an organization is focused on developing and integrating very large codes.

New modules are rapidly prototyped and then integrated into the large legacy codes. Finally, in the production workflow the goal is to create a deployed system and the development times may take years.

*IC design at advanced process nodes: Add flex to your flow*, Andrew Potemski, Synopsys, EDN, 8/16/2007.

This articles argues that carefully considering how your design flow is constructed greatly increases chances for success and that flexibility in a design flow is an important consideration as well.

*Metrics-Based Behavioral Design*, David Pursley, Forte Design Systems.

This paper dates to about 2006 and argues for a return to behavioral design as a way to improve productivity. It cites a study claiming a 50% productivity improvement from using behavioral design [Johnson98][Moussa98]. He introduces his idea of metrics-based behavioral design, and tries to develop a chain of metrics-based predictors to help his tools evaluate the value of each optimization. But, he only proposes a metric-based predictor for the original step of behavioral code evaluation (a tool to look at the original behavioral code and predict size/performance or even just whether the behavioral compiler could process it). In the end, he showed that there was some correlation between the CPU run time of his synthesis tool and his CDFG node count.

*Metrics Measure IC Design Productivity*, Michael Solka, Synopsys Inc.

This work was an EDA DesignLine paper found on the web. The main point was about how to gather data and what to gather as a project progresses to provide “actionable analyses of project practices and execution so that productivity improvement opportunities can be identified”. He suggests that metrics should be divided into 2 categories: “design characteristics” and “resource utilization”. Design characteristics are what you might expect: FET count, clock rate, lines of code, … Resource utilization has to do with the level of effort by personnel on the project and by CAD tool usage. He argues that both are important.

*Integrating FPGA IP Cores into a Topological Processing Environment*, Michael Vai and Jeremy Kepner.

This white paper argues for the use of reusable cores. It states that cores could be a big productivity booster. It then outlines a number of reasons why cores have not caught on: they are not well characterized enough to avoid repeated design spins and verification iterating, cores based on bitstreams are not portable, cores based on HDL's may be portable but if they are, they don't use FPGA-specific features and therefore are low performance, etc. Finally, the paper indicates that GFLOPS is not a particularly good metric, it is all about how the cores interface together that is important.

## A.2 List of Commercially Available High-Level FPGA Design Tools

We have identified a number of FPGA design tools that can be considered as “high-level” and listed them in the table below. The number of tools that are being introduced is growing and we will attempt to keep a current and accurate list during the course of the study. We will evaluate a number of these tools and summarize the others as part of this study.

| **Tool Name** | **Company** |
|---|---|
| DK Design Suite (Handel C), Agility Compiler | Celoxica |
| Pixel Streams, Hyper Streams | Celoxica |
| DSP Builder | Altera |
| System Generator | Xilinx |
| ImpulseC | ImpulseC |
| MitrionC | Mitrionics |
| AccelDSP | Xilinx |
| C2H | Altera |
| SynplifyDSP | Synplicity |
| Reconfigurable Computing Toolbox | DSPLogic |
| Simulink HDL Coder | Mathworks |
| Filter Design HDL Coder | Mathworks |
| Carte | SRC |
| CatapultC | Mentor |
| C2R | CebaTech |
| Cynthesizer | Forte |
| Computational Adrenaline | Concurrent EDA |
| Mobius | Codetronix |
| AutoPilot | AutoESL |
| Cascade CoProcessor | Critical Blue |
| LabView FPGA | National Instruments |
| BinaChip-FPGA | BinaChip |
| Bluespec SystemVerilog, Bluespec SystemC | Bluespec |
| Pico Express | Synfora |
| Dime-C | Nallatech |
| CoreFire | Annapolis Microsystems |
| Viva | Starbridge Systems |
| Stone Ridge Compiler Collection | Stone Ridge Technology |

## A.3 FPGA Architecture Survey

| Company | Niche |
|---|---|
| 3P plus 1 Technology | Coarse-grain configurable IP cores |
| Achronix Semiconductor Corp | Gigahertz asynchronous FPGA |
| Ambric | Massively parallel processor array with a structural object programming model |
| Ascenium Corp | In stealth mode |
| Aspex | Domain-specific system-on-chip with configurable IP cores |
| ChipWrights | RISC/SIMD/Vector processor architecture |
| Clearspeed | Low-power Floating point |
| Coherent Logix | Multi-core grid with allocatable interconnect |
| Context Corporation | Sweeney, Robertson, Tocher (MSDF/SRT) arithmetic processing algorithm and coarse-grained dynamic reconfigurability |
| Element CXI | An evolved QuickSilver Technologies with wrapped heterogeneous “elements” |
| Icera Semiconductor Ltd | Software-defined radios |
| Ikoa Corporation | Stealth mode - Memory-centric and defect-tolerant signal processing |
| Intellasys Corporation | Scalable Embedded Array Processor |
| IP Flex | Medium-grain reconfigurable fabric SoC closely coupled with RISC cores |
| M2000 | Used to be Meta systems, in stealth mode |
| MathStar | Field Programmable Object Area (FPOA) |
| Mesh Semiconductor | Stealth mode, or out of business |
| PACT | Coarse-grain ALU architecture |
| Picochip Designs | Multi-core processor array for signal processing |
| Rapport | An evolved PipeRench-like architecture |
| Raytheon | MONARCH architecture and development environment |
| ReCore | An array of specialized micro-sequenced processors |
| Sandbridge | Multi-core DSP arrays for comm applications |
| Spiral Gateway | Stealth mode |
| Stream Processors | VLIW-like processing engine |
| Stretch | Tightly integrated GPP with configurable fabric |
| Systemonic | Wireless networking acquired by Philips Semi |
| Tabula | Currently in stealth mode. |
| Silicon Hive | IP locks for comm and video applications |
| Videantis | Specialized video processing engines |
| Vivace Semiconductor | Specialized video processors |
| Xelerated | Configurable network processor |
| XMOS Semiconductor | Stealth mode |

## List of Acronyms, Abbreviations, and Symbols

| **Acronym** | **Description** |
|---|---|
| 3GL | Third generation languages |
| 4GL | Fourth generation languages |
| 5GL | Fifth generation languages |
| ART | Abstraction, Reuse, and Turns per day |
| CAD | Computer Aided Design |
| CCM | Configurable Computing Machine |
| CORBA | Common Object Request Broker Architecture |
| DSP | Digital Signal Processing |
| EDA | Electronic Design Automation |
| FDP | FPGA Design Productivity |
| FDTE | Formulation, Design, Translation, and Execution |
| FPGAs | Field Programmable Gate Arrays |
| HDLs | Hardware description languages |
| HPC | High-performance computing |
| HPEC | High-performance embedded computing |
| LLC | Low-level C |
| RTL | Register transfer level |
| SEI | Software Engineering Institute |
| SIRCA | Strategic Infrastructure for Reconfigurable Computing Applications |
| SOC | System-on-chip |
| SPC | Software Productivity Consortium |
| STAR | Special Technology Area Review |